Python\_Prob\_Stat\_Machine\_Learning\_Unpingco\_\_2E\_C03

124

3

Statistics

**3.1 Introduction**

To get started thinking about statistics, consider the three famous problems

* Suppose you have a bag ﬁlled with colored marbles. You close your eyes and reach into it and pull out a handful of marbles, what can you say about what is in the bag?
* You arrive in a strange town and you need a taxicab. You look out the window, and in the dark, you can just barely make out the number on the roof of one of the cabs. In this town, you know they label the cabs sequentially. How many cabs does the town have?
* You have already taken the entrance exam twice and you want to know if it’s worth it to take it a third time in the hopes that your score will improve. Because only the last score is reported, you are worried that you may do worse the third time. How do you decide whether or not to take the test again?

Statistics provides a structured way to approach each of these problems. This is important because it is easy to be fooled by your biases and intuitions. Unfortunately, the ﬁeld does not provide a single way to do this, which explains the many library shelves that groan under the weight of statistics texts. This means that although many statistical quantities are easy to compute, these are not so easy to justify, explain, or even understand. Fundamentally, when we start with just the data, we lack the underlying probability density that we discussed in the last chapter. This removes key structures that we have to compensate for in; however, we choose to process the data. In the following, we consider some of the most powerful statistical tools in the Python arsenal and suggest ways to think through them.

**3.2 Python Modules for Statistics**

**3.2.1 Scipy Statistics Module**

Although there are some basic statistical functions in Numpy (e.g., mean, std, median), the real repository for statistical functions is in scipy.stats. There are over eighty continuous probability distributions implemented in scipy.stats and an additional set of more than ten discrete distributions, along with many other supplementary statistical functions.

To get started with scipy.stats, you have to load the module and create an object that has the distribution you’re interested in. For example,

>>> import scipy.stats # might take awhile

>>> n = scipy.stats.norm(0,10) # create normal distrib

The n variable is an object that represents a normally distributed random variable with mean zero and standard deviation, σ = 10. Note that the more general term for these two parameters is location and scale, respectively. Now that we have this deﬁned, we can compute mean, as in the following:

>>> n.mean() # we already know this from its definition! 0.0

We can also compute higher order moments as

>>> n.moment(4) 30000.0 The main public methods for continuous random variables are

• rvs: random variates

• pdf: probability density function

• cdf: cumulative distribution function

• sf: survival Function (1-CDF)

• ppf: percent point function (Inverse of CDF)

• isf: inverse survival function (Inverse of SF)

• stats: mean, variance, (Fisher’s) skew, or (Fisher’s) kurtosis

• moment: non-central moments of the distribution

For example, we can compute the value of the pdf at a speciﬁc point.

>>> n.pdf(0) 0.03989422804014327 or, the cdf for the same random variable.

>>> n.cdf(0) 0.5 3.2 Python Modules for Statistics

125

You can also create samples from this distribution as in the following:

>>> n.rvs(10) array([15.3244518 , -9.4087413 , 6.94760096, 6.9753351 , 7.95314387, -3.18127815,

0.61627683, -3.92073633,

5.69087949, 0.84197674])

Many common statistical tests are already built-in. For example, Shapiro–Wilks tests the null hypothesis that the data were drawn from a normal distribution, 1 as in the following:

>>> scipy.stats.shapiro(n.rvs(100)) (0.9749656915664673, 0.05362436920404434)

The second value in the tuple is the p-value (discussed below).

3.2.2 Sympy Statistics Module

Sympy has its own much smaller, but still extremely useful statistics module that enables symbolic manipulation of statistical quantities. For example,

>>> from sympy import stats, sqrt, exp, pi

>>> X = stats.Normal('x',0,10) # create normal random variable

We can obtain the probability density function as

>>> from sympy.abc import x

>>> stats.density(X)(x)

sqrt(2)\*exp(-x\*\*2/200)/(20\*sqrt(pi))

>>> sqrt(2)\*exp(-x\*\*2/200)/(20\*sqrt(pi)) sqrt(2)\*exp(-x\*\*2/200)/(20\*sqrt(pi))

and we can evaluate the cumulative density function as in the following:

>>> stats.cdf(X)(0) 1/2

Note that you can evaluate this numerically by using the evalf() method on the output. Sympy provides intuitive ways to consider standard probability questions by using the stats.P function, as in the following:

>>> stats.P(X>0) # prob X >0? 1/2

There is also a corresponding expectation function, stats.E you can use to compute complicated expectations using all of Sympy’s powerful built-in integration machinery. For example we can compute, E( √| X | ) in the following:

>>> stats.E(abs(X)\*\*(1/2)).evalf() 2.59995815363879

Unfortunately, there is very limited support for multivariate distributions at the time of this writing.

1 We will explain null hypothesis and the rest of it later. 126

3

Statistics

3.2.3 Other Python Modules for Statistics

There are many other important Python modules for statistical work. Two important modules are Seaborn and Statsmodels. As we discussed earlier, Seaborn is library built on top of Matplotlib for very detailed and expressive statistical visualizations, ideally suited for exploratory data analysis. Statsmodels is designed to complement Scipy with descriptive statistics, estimation, and inference for a large variety of statistical models. Statsmodels includes (among many others) generalized linear models, robust linear models, and methods for time-series analysis, with an emphasis on econometric data and problems. Both these modules are well supported and very well documented and designed to integrate tightly into Matplotlib, Numpy, Scipy, and the rest of the scientiﬁc Python stack. Because the focus of this text is more conceptual as opposed to domain speciﬁc, I have chosen not to emphasize either of these, notwithstanding how powerful each is.

3.3 Types of Convergence

The absence of the probability density for the raw data means that we have to argue about sequences of random variables in a structured way. From basic calculus, recall the following convergence notation:

x n → xo

for the real number sequence x n . This means that for any given > 0, no matter how small, we can exhibit a m such that for any n > m, we have

| x n − x o | <

Intuitively, this means that once we get past m in the sequence, we get as to within • of x o . This means that nothing surprising happens in the sequence on the long march to inﬁnity, which gives a sense of uniformity to the convergence process. When we argue about convergence for statistics, we want to same look-and-feel as we have here, but because we are now talking about random variables, we need other concepts. There are two moving parts for random variables. Recall from our probability chapter that random variables are really functions that map sets into the real line: X : Ω ↦ → R. Thus, one part is the behavior of the subsets of Ω in terms of convergence. The other part is how the sequences of real values of the random variable behave in convergence.

3.3.1 Almost Sure Convergence

The most straightforward extension into statistics of this convergence concept is almost sure convergence, which is also known as convergence with probability one, 3.3 Types of Convergence

127

P { for each > 0 there is n > 0 such that for all n > n , | X n − X | < } = 1

(3.3.1.1) Note the similarity to the prior notion of convergence for real numbers. When this as happens, we write this as X n → X. In this context, almost sure convergence means that if we take any particular ω ∈ Ω and then look at the sequence of real numbers that are produced by each of the random variables,

(X 1 (ω), X 2 (ω), X 3 (ω), . . . , X n (ω))

then this sequence is just a real-valued sequence in the sense of our convergence on the real line and converges in the same way. If we collect all of the ω for which this is true and the measure of that collection equals one, then we have almost sure convergence of the random variable. Notice how the convergence idea applies to both sides of the random variable: the (domain) Ω side and the (co-domain) real-valued side.

An equivalent and more compact way of writing this is the following:

P X n (ω) = X(ω) Ω lim = 1 ω : ∈ ( n→∞ )

Example. To get some feel for the mechanics of this kind of convergence consider the following sequence of uniformly distributed random variables on the unit interval, X n ∼ U [ 0, 1 ] . Now, consider taking the maximum of the set of n such variables as the following:

X (n) = max { X 1 , . . . , X n }

In other words, we scan through a list of n uniformly distributed random variables and pick out the maximum over the set. Intuitively, we should expect that X (n) should somehow converge to one. Let’s see if we can make this happen almost surely. We want to exhibit m so that the following is true,

P( | 1 − X (n) | ) < when n > m

Because X (n) < 1, we can simplify this as the following:

1 − P(X (n) < ) = 1 − (1 − ) m −→ 1 m→∞

Thus, this sequence converges almost surely. We can work this example out in Python using Scipy to make it concrete with the following code:

>>> from scipy import stats

>>> u=stats.uniform()

>>> xn = lambda i: u.rvs(i).max()

>>> xn(5)

0.966717838482003 128

3

Statistics

Thus, the xn variable is the same as the X (n) random variable in our example. Figure3.1 shows a plot of these random variables for different values of n and multiple realizations of each random variable (multiple gray lines). The dark horizontal line is at the 0.95 level. For this example, suppose we are interested in the convergence of the random variable to within 0.05 of one so we are interested in the region between one and 0.95. Thus, in our Eq.3.3.1.1, = 0.05. Now, we have to ﬁnd n to get the almost sure convergence. From Fig.3.1, as soon as we get past n > 60, we can see that all the realizations start to ﬁt in the region above the 0.95 horizontal line. However, there are still some cases where a particular realization will skip below this line. To get the probability guarantee of the deﬁnition satisﬁed, we have to make sure that for whatever n we settle on, the probability of this kind of noncompliant behavior should be extremely small, say, less than 1%. Now, we can compute the following to estimate this probability for n = 60 over 1000 realizations:

>>> import numpy as np

>>> np.mean([xn(60) > 0.95 for i in range(1000)])

0.961

So, the probability of having a noncompliant case beyond n > 60 is pretty good, but not still what we are after (0.99). We can solve for the m in our analytic proof of convergence by plugging in our factors for and our desired probability constraint,

>>> print (np.log(1-.99)/np.log(.95)) 89.78113496070968

Now, rounding this up and re-visiting the same estimate as above,

>>> import numpy as np

>>> np.mean([xn(90) > 0.95 for i in range(1000)])

0.995

which is the result we were looking for. The important thing to understand from this example is that we had to choose convergence criteria for both the values of

Fig. 3.1 Almost sure convergence example for multiple realizations of the limiting sequence 3.3 Types of Convergence

129

the random variable (0.95) and for the probability of achieving that level (0.99) in order to compute the m. Informally speaking, almost sure convergence means that not only will any particular X n be close to X for large n, but whole sequence of values will remain close to X with high probability.

3.3.2 Convergence in Probability

A weaker kind of convergence is convergence in probability which means the following:

P( | X n − X | > ) → 0

as n → ∞ for each > 0.

P

This is notationally shown as X n → X. For example, let’s consider the following sequence of random variables where X n = 1/2 n with probability p n and where

P

X n = c with probability 1 − p n . Then, we have X n → 0 as p n → 1. This is allowable under this notion of convergence because a diminishing amount of nonconverging behavior (namely, when X n = c) is possible. Note that we have said nothing about how p n → 1.

Example. To get some sense of the mechanics of this kind of convergence, let { X 1 , X 2 , X 3 , . . . } be the indicators of the corresponding intervals,

(0, 1 ] , (0, 2 1 ] , ( 2 1 , 1 ] , (0, 3 1 ] , ( 3 1 , 2 3 ] , ( 2 3 , 1 ]

In other words, just keep splitting the unit interval into equal chunks and enumerate those chunks with X i . Because each X i is an indicator function, it takes only two values: zero and one. For example, for X 2 = 1 if 0 < x ≤ 1/2 and zero otherwise. Note that x ∼ U(0, 1). This means that P(X 2 = 1) = 1/2. Now, we want to compute the sequence of P(X n > ) for each n for some ∈ (0, 1). For X 1 , we have P(X 1 > ) = 1 because we already chose in the interval covered by X 1 . For X 2 , we have P(X 2 > ) = 1/2, for X 3 , we have P(X 3 > ) = 1/3, and so on. This produces the following sequence: (1, 2 1 , 2 1 , 3 1 , 3 1 , . . .). The limit of the

P

sequence is zero so that X n → 0. However, for every x ∈ (0, 1), the sequence of function values of X n (x) consists of inﬁnitely many zeros and ones (remember that indicator functions can evaluate to either zero or one). Thus, the set of x for which the sequence X n (x) converges is empty because the sequence bounces between zero and one. This means that almost sure convergence fails here even though we have convergence in probability. The key distinction is that convergence in probability considers the convergence of a sequence of probabilities whereas almost sure convergence is concerned about the sequence of values of the random variables over sets of events that ﬁll out the underlying probability space entirely (i.e., with probability one).

This is a good example so let’s see if we can make it concrete with some Python. The following is a function to compute the different subintervals: 130

3

Statistics

>>> make\_interval= lambda n: np.array(list(zip(range(n+1),

...

range(1,n+1))))/n

Now, we can use this function to create a Numpy array of intervals, as in the example,

>>> intervals= np.vstack([make\_interval(i) for i in range(1,5)])

>>> print (intervals) [[0. 1. ] [0. 0.5 ] [0.5 1. ] [0. 0.33333333] [0.33333333 0.66666667] [0.66666667 1. ] [0. 0.25 ] [0.25 0.5 ] [0.5 0.75 ] [0.75 1. ]]

The following function computes the bit string in our example, { X 1 , X 2 , . . . , X n } :

>>> bits= lambda u:((intervals[:,0] < u) & (u<=intervals[:,1])).astype(int)

>>> bits(u.rvs())

array([1, 0, 1, 0, 0, 1, 0, 0, 0, 1])

Now that we have the individual bit strings, to show convergence we want to show that the probability of each entry goes to a limit. For example, using ten realizations,

>>> print (np.vstack([bits(u.rvs()) for i in range(10)])) [[1 1 0 1 0 0 0 1 0 0] [1 1 0 1 0 0 0 1 0 0] [1 1 0 0 1 0 0 1 0 0] [1 0 1 0 0 1 0 0 1 0] [1 0 1 0 0 1 0 0 1 0] [1 1 0 0 1 0 0 1 0 0] [1 1 0 1 0 0 1 0 0 0] [1 1 0 0 1 0 0 1 0 0] [1 1 0 0 1 0 0 1 0 0] [1 1 0 1 0 0 1 0 0 0]]

We want the limiting probability of a one in each column to convert to a limit. We can estimate this over 1000 realizations using the following code:

>>> np.vstack([bits(u.rvs()) for i in range(1000)]).mean(axis=0) array([1. , 0.493, 0.507, 0.325, 0.34 , 0.335, 0.253, 0.24 , 0.248, 0.259])

Note that these entries should approach the (1, 2 1 , 2 1 , 3 1 , 3 1 , . . .) sequence we found earlier. Figure3.2 shows the convergence of these probabilities for a large number of intervals. Eventually, the probability shown on this graph will decrease to zero 3.3 Types of Convergence

131

Fig. 3.2 Convergence in probability for the random variable sequence

with large enough n. Again, note that the individual sequences of zeros and ones do not converge, but the probabilities of these sequences converge. This is the key difference between almost sure convergence and convergence in probability. Thus, convergence in probability does not imply almost sure convergence. Conversely, almost sure convergence does imply convergence in probability.

The following notation should help emphasize the difference between almost sure convergence and convergence in probability, respectively,

P | X n − X | < lim = 1(almost sure convergence) ( n→∞ )

lim P( | X n − X | < ) = 1(convergence in probability) n→∞

3.3.3 Convergence in Distribution

So far, we have been discussing convergence in terms of sequences of probabilities or sequences of values taken by the random variable. By contrast, the next major kind of convergence is convergence in distribution where

lim F n (t) = F(t) n→∞

for all t for which F is continuous and F is the cumulative density function. For this case, convergence is only concerned with the cumulative density function, written

d as X n → X.

Example. To develop some intuition about this kind of convergence, consider a sequence of X n Bernoulli random variables. Furthermore, suppose these are all really

d

just the same random variable X. Trivially, Xn

→ X. Now, suppose we deﬁne

d

Y = 1 − X, which means that Y has the same distribution as X. Thus, X n → Y. By 132

3

Statistics

contrast, because | X n −Y | = 1 for all n, we can never have almost sure convergence or convergence in probability. Thus, convergence in distribution is the weakest of the three forms of convergence in the sense that it is implied by the other two, but implies neither of the two.

d

→ Z where Z ∼ N(0, 1), but d we could also have Y n → −Z. That is, Y n could converge in distribution to either Z or −Z. This may seem ambiguous, but this kind of convergence is practically very useful because it allows for complicated distributions to be approximated by simpler distributions.

As another striking example, we could have Yn

3.3.4 Limit Theorems

Now that we have all of these notions of convergence, we can apply them to different situations and see what kinds of claims we can construct from them.

Weak Law of Large Numbers. Let { X 1 , X 2 , . . . , X n } be an iid (independent, identically distributed) set of random variables with ﬁnite mean E(X k ) = μ and ﬁnite

P

1 variance. Let X n = n ∑ k X k . Then, we have X n → μ . This result is important because we frequently estimate parameters using an averaging process of some kind. This basically justiﬁes this in terms of convergence in probability. Informally, this means that the distribution of X n becomes concentrated around μ as n → ∞.

Strong Law of Large Numbers. Let { X 1 , X 2 , . . . , } be an iid set of random varias E ables. Suppose that μ = | X i | < ∞, then X n → μ . The reason this is called the strong law is that it implies the weak law because almost sure convergence implies convergenceinprobability. Theso-calledKomogorovcriteriongives theconvergence of the following:

2 σ k ∑ k2 k

as a sufﬁcient condition for concluding that the Strong Law applies to the sequence { X k } with corresponding { σ k 2 } . As an example, consider an inﬁnite sequence of Bernoulli trials with X i = 1 if the i th trial is successful. Then X n is the relative frequency of successes in n trials and E(X i ) is the probability p of success on the i th trial. With all that established, the Weak Law says only that if we consider a sufﬁciently large and ﬁxed n, the probability that the relative frequency will converge to p is guaranteed. The Strong Law states that if we regard the observation of all the inﬁnite { X i } as one performance of the experiment, the relative frequency of successes will almost surely converge to p. The difference between the Strong Law and the Weak Law of large numbers is subtle and rarely arises in practical applications of probability theory. 3.3 Types of Convergence

133

Central Limit Theorem. Although the Weak Law of Large Numbers tells us that the distribution of X n becomes concentrated around μ , it does not tell us what that distribution is. The central limit theorem (CLT) says that X n has a distribution that is approximately Normal with mean μ and variance σ 2 /n. Amazingly, nothing is assumed about the distribution of X i , except the existence of the mean and variance. The following is the Central Limit Theorem: Let { X 1 , X 2 , . . . , X n } be iid with mean μ and variance σ 2 . Then,

n(Xn − μ ) P Z n = √ −→ Z ∼ N(0, 1)

σ

The loose interpretation of the Central Limit Theorem is that X n can be legitimately approximated by a Normal distribution. Because we are talking about convergence in probability here, claims about probability are legitimized, not claims about the random variable itself. Intuitively, this shows that normality arises from sums of small, independent disturbances of ﬁnite variance. Technically, the ﬁnite variance assumption is essential for normality. Although the Central Limit Theorem provides a powerful, general approximation, the quality of the approximation for a particular situation still depends on the original (usually unknown) distribution.

3.4 Estimation Using Maximum Likelihood

The estimation problem starts with the desire to infer something meaningful from data. For parametric estimation, the strategy is to postulate a model for the data and then use the data to ﬁt model parameters. This leads to two fundamental questions: where to get the model and how to estimate the parameters? The ﬁrst question is best answered by the maxim: all models are wrong, some are useful. In other words, choosing a model depends as much on the application as on the model itself. Think about models as building different telescopes to view the sky. No one would ever claim that the telescope generates the sky! It is same with data models. Models give us multiple perspectives on the data that themselves are proxies for some deeper underlying phenomenon.

Some categories of data may be more commonly studied using certain types of models, but this is usually very domain speciﬁc and ultimately depends on the aims of the analysis. In some cases, there may be strong physical reasons behind choosing a model. For example, one could postulate that the model is linear with some noise as in the following:

Y = aX +

which basically says that you, as the experimenter, dial in some value for X and then read off something directly proportional to X as the measurement, Y, plus some additive noise that you attribute to jitter in the apparatus. Then, the next step is to estimate the parameter a in the model, given some postulated claim about the nature 134

3

Statistics

of . How to compute the model parameters depends on the particular methodology. The two broad rubrics are parametric and nonparametric estimation. In the former, we assume we know the density function of the data and then try to derive the embedded parameters for it. In the latter, we claim only to know that the density function is a member of a broad class of density functions and then use the data to characterize a member of that class. Broadly speaking, the former consumes less data than the latter, because there are fewer unknowns to compute from the data.

Let’s concentrate on parametric estimation for now. The tradition is to denote the unknown parameter to be estimated as θ which is a member of a large space of alternates, Θ. To judge between potential θ values, we need an objective function, ˆ ˆ known as a risk function, L(θ, θ), where θ(x) is an estimate for the unknown θ that is derived from the available data x. The most common and useful risk function is the squared error loss,

ˆ L(θ, θ) ˆ = (θ − θ)2

Although neat, this is not practical because we need to know the unknown θ to ˆ compute it. The other problem is because θ is a function of the observed data, it is also a random variable with its own probability density function. This leads to the notion of the expected risk function,

R(θ, θ) ˆ = E θ (L(θ, ˆ θ)) = L(θ, ˆ θ(x)) f (x ; θ)dx ∫

In other words, given a ﬁxed θ, integrate over the probability density function of the data, f (x), to compute the risk. Plugging in for the squared error loss, we compute the mean squared error,

E θ (θ − θ) 2 ˆ = (θ − θ) 2 ˆ f (x ; θ)dx ∫

This has the important factorization into the bias,

bias = E θ ( θ) ˆ − θ

ˆ with the corresponding variance, V θ ( θ) as in the following mean squared error (MSE):

ˆ E θ (θ − θ) 2 ˆ = bias 2 + V θ ( θ)

This is an important trade-off that we will return to repeatedly. The idea is the bias is ˆ nonzero when the estimator θ, integrated over all possible data, f (x), does not equal the underlying target parameter θ. In some sense, the estimator misses the target, no matter how much data is used. When the bias equals zero, the estimated is unbiased. For ﬁxed MSE, low bias implies high variance and vice versa. This trade-off was once not emphasized and instead much attention was paid to the smallest variance of unbiased estimators (see Cramer–Rao bounds). In practice, understanding and 3.4 Estimation Using Maximum Likelihood

135

exploiting the trade-off between bias and variance and reducing the MSE is more important.

With all this setup, we can now ask how bad can bad get by examining minimax risk,

ˆ R mmx = inf sup R(θ, θ)

ˆ θ

θ

where the inf is take over all estimators. Intuitively, this means if we found the ˆ worst possible θ and swept over all possible parameter estimators θ, and then took the smallest possible risk we could ﬁnd, we would have the minimax risk. Thus, an ˆ estimator, θ mmx , is a minimax estimator if it achieves this feat,

ˆ sup R(θ, θ mmx ˆ ) = inf sup R(θ, θ) ˆ θ θ θ

ˆ In other words, even in the face of the worst θ (i.e., the sup θ ), θ mmx still achieves the minimax risk. There is a greater theory that revolves around minimax estimators of various kinds, but this is far beyond our scope here. The main thing to focus on is that under certain technical but easily satisﬁable conditions, the maximum likelihood estimator is approximately minimax. Maximum likelihood is the subject of the next section. Let’s get started with the simplest application: coin-ﬂipping.

3.4.1 Setting Up the Coin-Flipping Experiment

Suppose we have coin and want to estimate the probability of heads (p) for it. We model the distribution of heads and tails as a Bernoulli distribution with the following probability mass function:

φ (x) = p x (1 − p)(1−x)

where x is the outcome, 1 for heads and 0 for tails. Note that maximum likelihood is a parametric method that requires the speciﬁcation of a particular model for which we will compute embedded parameters. For n independent ﬂips, we have the joint density as the product of n of these functions as in,

n φ (x) = ∏ p i x (1 − p)(1−x i ) i=1

The following is the likelihood function:

n L(p ; x) = ∏ p x i (1 − p)1−x i i=1 136

3

Statistics

This is basically notation. We have just renamed the previous equation to emphasize the p parameter, which is what we want to estimate.

The principle of maximum likelihood is to maximize the likelihood as the function of p after plugging in all of the x i data. We then call this maximizer pˆ which is a function of the observed x i data, and as such, is a random variable with its own distribution. This method therefore ingests data and an assumed model for the probability density, and produces a function that estimates the embedded parameter in the assumed probability density. Thus, maximum likelihood generates the functions of data that we need in order to get at the underlying parameters of the model. Note that there is no limit to the ways we can functionally manipulate the data we have collected. The maximum likelihood principle gives us a systematic method for constructing these functions subject to the assumed model. This is a point worth emphasizing: the maximum likelihood principle yields functions as solutions the same way solving differential equations yields functions as solutions. It is very, very much harder to produce a function than to produce a value as a solution, even with the assumption of a convenient probability density. Thus, the power of the principle is that you can construct such functions subject to the model assumptions.

Simulating the Experiment. We need the following code to simulate coin-ﬂipping:

>>> from scipy.stats import bernoulli

>>> p\_true=1/2.0 # estimate this!

>>> fp=bernoulli(p\_true) # create bernoulli random variate

>>> xs = fp.rvs(100) # generate some samples

>>> print (xs[:30]) # see first 30 samples

[0 1 0 1 1 0 0 1 1 1 0 1 1 1 0 1 1 0 1 1 0 1 0 0 1 1 0 1 0 1]

Now, we can write out the likelihood function using Sympy. Note that we give the Sympy variables the positive=True attribute upon construction because this eases Sympy’s internal simpliﬁcation algorithms.

>>> import sympy

>>> x,p,z=sympy.symbols('x p z', positive=True)

>>> phi=p\*\*x\*(1-p)\*\*(1-x) # distribution function

>>> L=np.prod([phi.subs(x,i) for i in xs]) # likelihood function

>>> print (L) # approx 0.5? p\*\*57\*(-p + 1)\*\*43

Note that, once we plug in the data, the likelihood function is solely a function of the unknown parameter (p in this case). The following code uses calculus to ﬁnd the extrema of the likelihood function. Note that taking the log of L makes the maximization problem tractable but doesn’t change the extrema.

>>> logL=sympy.expand\_log(sympy.log(L))

>>> sol,=sympy.solve(sympy.diff(logL,p),p)

>>> print (sol) 57/100 3.4 Estimation Using Maximum Likelihood

137

Programming Tip

Note that sol,=sympy.solve statement includes a comma after the sol variable. This is because the solve function returns a list containing a single element. Using this assignment unpacks that single element into the sol variable directly. This is another one of the many small elegancies of Python.

The following code generates Fig.3.3.

fig,ax=subplots() x=np.linspace(0,1,100) ax.plot(x,map(sympy.lambdify(p,logJ,'numpy'),x),'k-',lw=3) ax.plot(sol,logJ.subs(p,sol),'o',

color='gray',ms=15,label='Estimated') ax.plot(p\_true,logJ.subs(p,p\_true),'s',

color='k',ms=15,label='Actual') ax.set\_xlabel('$p$',fontsize=18) ax.set\_ylabel('Likelihood',fontsize=18) ax.set\_title('Estimate not equal to true value',fontsize=18) ax.legend(loc=0)

Programming Tip

In the prior code, we use the lambdify function in lambdify(p,logJ, ’numpy’) to take a Sympy expression and convert it into a Numpy version that is easier to compute. The lambdify function has an extra argument where you can specify the function space that it should use to convert the expression. In the above this is set to Numpy.

Figure3.3 shows that our estimator pˆ (circle) is not equal to the true value of p (square), despite being the maximum of the likelihood function. This may sound disturbing, but keep in mind this estimate is a function of the random data; and since that data can change, the ultimate estimate can likewise change. Remember that the estimator is a function of the data and is thus also a random variable, just like the data is. This means it has its own probability distribution with corresponding mean and variance. So, what we are observing is a consequence of that variance.

Figure3.4 shows what happens when you run many thousands of coin experiments and compute the maximum likelihood estimate for each experiment, given a particular number of samples per experiment. This simulation gives us a histogram of the maximum likelihood estimates, which is an approximation of the probability distribution of the pˆ estimator itself. This ﬁgure shows that the sample mean of the 138

3

Statistics

Fig. 3.3 Maximum likelihood estimate versus true parameter. Note that the estimate is slightly off from the true value. This is a consequence of the fact that the estimator is a function of the data and lacks knowledge of the true underlying value

Fig. 3.4 Histogram of maximum likelihood estimates. The title shows the estimated mean and standard deviation of the samples

1 estimator ( μ = n ∑ pi ˆ ) is pretty close to the true value, but looks can be deceiving. The only way to know for sure is to check if the estimator is unbiased, namely, if

E( p)ˆ = p

Because this problem is simple, we can solve for this in general noting that the terms above are either p, if x i = 1 or 1 − p if x i = 0. This means that we can write

L(p | x) = p ∑ i=1 n x i (1 − p)n− ∑ i=1 n x i 3.4 Estimation Using Maximum Likelihood

139

with corresponding logarithm as

n n J = log(L(p | x)) = log(p) ∑ x i + log(1 − p) n − ∑ xi i=1 ( i=1 )

Taking the derivative of this gives

n (n dJ 1 xi ) = x i + ∑i=1 n dp p ∑ p − 1 i=1

and solving this for p leads to

n 1 pˆ = ∑ xi n i=1

This is our estimator for p. Up until now, we have been using Sympy to solve for this based on the data x i but now that we have it analytically we don’t have to solve for it each time. To check if this estimator is biased, we compute its expectation:

n 1 1 E ( pˆ ) = ∑ E(x i ) = ) nE(xi n n i

by linearity of the expectation and where

E(x i ) = p

Therefore,

E ( pˆ ) = p

This means that the estimator is unbiased. Similarly,

n 2 1 ⎡ ⎤ E E xi p2 = ( ) n 2 ∑ ⎣ ( i=1 ) ⎦

and where

E xi 2 = p ( )

and by the independence assumption,

E ( x i x j ) = E(x i )E(x j ) = p2 140

3

Statistics

Thus,

1 E n p2 = p (n 1)p2 + ( ) ( n 2 ) [ ]

So, the variance of the estimator, p,ˆ is the following:

p(1 − p) p2 E V( p)ˆ = E ( p ) 2 = ( ) n

Note that the n in the denominator means that the variance asymptotically goes to zero as n increases (i.e., we consider more and more samples). This is good news because it means that more and more coin-ﬂips lead to a better estimate of the underlying p.

Unfortunately, this formula for the variance is practically useless because we need p to compute it and p is the parameter we are trying to estimate in the ﬁrst place! However, this is where the plug-in principle 2 saves the day. It turns out in this situation, you can simply substitute the maximum likelihood estimator, p,ˆ for the p in the above equation to obtain the asymptotic variance for V( ˆp). The fact that this work is guaranteed by the asymptotic theory of maximum likelihood estimators.

Nevertheless, looking at V( p)2 ˆ , we can immediately notice that if p = 0, then there is no estimator variance because the outcomes are guaranteed to be tails. Also, for any n, the maximum of this variance happens at p = 1/2. This is our worst-case scenario and the only way to compensate is with larger n.

All we have computed is the mean and variance of the estimator. In general, this is insufﬁcient to characterize the underlying probability density of p,ˆ except if we somehow knew that pˆ were normally distributed. This is where the powerful Central Limit Theorem we discussed in Sect.3.3.4 comes in. The form of the estimator, which is just a sample mean, implies that we can apply this theorem and conclude that pˆ is asymptotically normally distributed. However, it doesn’t quantify how many samples n we need. In our simulation this is no problem because we can generate as much data as we like, but in the real world, with a costly experiment, each sample may be precious. 3 In the following, we won’t apply the Central Limit Theorem and instead proceed analytically.

Probability Density for the Estimator. To write out the full density for p,ˆ we ﬁrst have to ask what is the probability that the estimator will equal a speciﬁc value and the tally up all the ways that could happen with their corresponding probabilities. For example, what is the probability that

2 This is also known as the invariance property of maximum likelihood estimators. It basically states that the maximum likelihood estimator of any function, say, h(θ), is the same h with the maximum likelihood estimator for θ substituted in for θ; namely, h(θ ML ).

3 It turns out that the central limit theorem augmented with an Edgeworth expansion tells us that convergence is regulated by the skewness of the distribution [1]. In other words, the more symmetric the distribution, the faster it converges to the normal distribution according to the central limit theorem. 3.4 Estimation Using Maximum Likelihood

141

n 1 pˆ = ∑ x i = 0 n i=1

This can only happen one way: when x i = 0 ∀i. The probability of this happening can be computed from the density

n f (x, p) = ∏ (1 − p)1−x i px i ( ) i=1

n f ∑ x i = 0, p = (1 − p)n ( i=1 )

Likewise, if { x i } has only one nonzero element, then

n n−1 f ∑ x i = 1, p = np ∏ (1 − p) ( i=1 ) i=1

where the n comes from the n ways to pick one element from the n elements x i . Continuing this way, we can construct the entire density as

n n f ∑ x i = k, p = ( k ) p k (1 − p)n−k ( i=1 )

where the ﬁrst term on the right is the binomial coefﬁcient of n things taken k at a time. This is the binomial distribution and it’s not the density for p,ˆ but rather for n p.ˆ We’ll leave this as-is because it’s easier to work with below. We just have to remember to keep track of the n factor.

Conﬁdence Intervals. Now that we have the full density for p,ˆ we are ready to ask some meaningful questions. For example, what is the probability the estimator is within fraction of the true value of p?

P ( | pˆ − p | ≤ p )

More concretely, we want to know how often the estimated pˆ is trapped within of the actual value. That is, suppose we ran the experiment 1000 times to generate 1000 different estimates of p.ˆ What percentage of the 1000 so-computed values are trapped within of the underlying value. Rewriting the above equation as the following:

n P ( p − p < pˆ < p + p ) = P np − np < ∑ x i < np + np ( i=1 ) 142

3

Statistics

Let’spluginsomelivenumbershereforourworst-casescenario(i.e.,highestvariance scenario) where p = 1/2. Then, if = 1/100, we have

n 99n 101n P ∑ x i < < 100 100 ( i=1 )

Since the sum in integer valued, we need n > 100 to even compute this. Thus, if n = 101 we have,

101 101 9999 10201 P ∑ x i < = f ∑ x i = 50, p . . . < 200 200 ( i=1 ) ( i=1 )

101 = (1 − 1/2) 101−50 = 0.079 (1/2)50 ( 50 )

This means that in the worst-case scenario for p = 1/2, given n = 101 trials, we will only get within 1% of the actual p = 1/2 about 8% of the time. If you feel disappointed, it is because you’ve been paying attention. What if the coin was really heavy and it was hard work to repeat this 101 times?

Let’s come at this another way: given I could only ﬂip the coin 100 times, how close could I come to the true underlying value with high probability (say, 95%)? In this case, instead of picking a value for , we are solving for . Plugging in gives

100 P 50 − 50 < ∑ x i < 50 + 50 = 0.95 ( i=1 )

which we have to solve for . Fortunately, all the tools we need to solve for this are already in Scipy

>>> from scipy.stats import binom

>>> # n=100, p = 0.5, distribution of the estimator phat

>>> b=binom(100,.5)

>>> # symmetric sum the probability around the mean

>>> g = lambda i:b.pmf(np.arange(-i,i)+50).sum()

>>> print (g(10)) # approx 0.95

0.9539559330706295

The two vertical lines in Fig.3.5 show how far out from the mean we have to go to accumulate 95% of the probability. Now, we can solve this as

50 + 50 = 60 3.4 Estimation Using Maximum Likelihood

143

Fig. 3.5 Probability mass function for p.ˆ The two vertical lines form the conﬁdence interval

which makes = 1/5 or 20%. So, ﬂipping 100 times means I can only get within 20% of the real p 95% of the time in the worst-case scenario (i.e., p = 1/2). The following code veriﬁes the situation:

>>> from scipy.stats import bernoulli

>>> b=bernoulli(0.5) # coin distribution

>>> xs = b.rvs(100) # flip it 100 times

>>> phat = np.mean(xs) # estimated p

>>> print (abs(phat-0.5) < 0.5\*0.20) # make it w/in interval? True

Let’s keep doing this and see if we can get within this interval 95% of the time.

>>> out=[]

>>> b=bernoulli(0.5) # coin distribution

>>> for i in range(500): # number of tries ... xs = b.rvs(100) # flip it 100 times ... phat = np.mean(xs) # estimated p ... out.append(abs(phat-0.5) < 0.5\*0.20 ) # within 20% ? ...

>>> # percentage of tries w/in 20% interval

>>> print (100\*np.mean(out))

97.39999999999999

Well, that seems to work! Now we have a way to get at the quality of the estimator, p.ˆ

Maximum Likelihood Estimator Without Calculus. The prior example showed howwecanusecalculus tocomputethemaximumlikelihoodestimator. It’s important to emphasize that the maximum likelihood principle does not depend on calculus and extends to more general situations where calculus is impossible. For example, let X be uniformly distributed in the interval [ 0, θ ] . Given n measurements of X, the likelihood function is the following:

n 1 1 L(θ) = ∏ θ = θn i=1 144

3

Statistics

where each x i ∈ [ 0, θ ] . Note that the slope of this function is not zero anywhere so the usual calculus approach is not going to work here. Because the likelihood is the product of the individual uniform densities, if any of the x i values were outside of the proposed [ 0, θ ] interval, then the likelihood would go to zero, because the uniform density is zero outside of the [ 0, θ ] . This is no good for maximization. Thus, observing that the likelihood function is strictly decreasing with increasing θ, we conclude that the value for θ that maximizes the likelihood is the maximum of the x i values. To summarize, the maximum likelihood estimator is the following:

θ ML = max xi i

As always, we want the distribution of this estimator to judge its performance. In this case, this is pretty straightforward. The cumulative density function for the max function is the following:

P θ ML ˆ < v ≤ v ∧ x 1 ≤ v . . . ∧ x n ≤ v) = P(x0 ( )

and since all the x i are uniformly distributed in [ 0, θ ] , we have

v ˆ P θ ML <v = ( ) ( θ )n

So, the probability density function is then,

f θ ML ˆ (θ ML ) = nθ ML n−1 θ−n

Then, we can compute the E(θ ML ) = (θn)/(n + 1) with corresponding variance as V(θ ML ) = (θ 2 n)/(n + 1) 2 /(n + 2).

For a quick sanity check, we can write the following simulation for θ = 1 as in the following:

>>> from scipy import stats

>>> rv = stats.uniform(0,1) # define uniform random variable

>>> mle=rv.rvs((100,500)).max(0) # max along row-dimension

>>> print (mean(mle)) # approx n/(n+1) = 100/101 ˜= 0.99

0.989942138048

>>> print (var(mle)) #approx n/(n+1)\*\*2/(n+2) ˜= 9.61E-5

9.95762009884e-05

Programming Tip

The max(0) sufﬁx on for the mle computation takes the maximum of the socomputed array along the row (axis=0) dimension. 3.4 Estimation Using Maximum Likelihood

145

You can also plot hist(mle) to see the histogram of the simulated maximum likelihood estimates and match it up against the probability density function we derived above.

In this section, we explored the concept of maximum likelihood estimation using a coin-ﬂipping experiment both analytically and numerically with the scientiﬁc Python stack. We also explored the case when calculus is not workable for maximum likelihood estimation. There are two key points to remember. First, maximum likelihood estimation produces a function of the data that is itself a random variable, with its own probability distribution. We can get at the quality of the so-derived estimators by examining the conﬁdence intervals around the estimated values using the probability distributions associated with the estimators themselves. Second, maximum likelihood estimation applies even in situations where using basic calculus is not applicable [2].

3.4.2 Delta Method

Sometimes we want to characterize the distribution of a function of a random variable. In order to extend and generalize the Central Limit Theorem in this way, we need the Taylor series expansion. Recall that the Taylor series expansion is an approximation of a function of the following form:

r g (i) (a) T r ∑ i! (x(x) −= a)i i=0

this basically says that a function g can be adequately approximated about a point a using a polynomial based on its derivatives evaluated at a. Before we state the general theorem, let’s examine an example to understand how the mechanics work.

Example. Suppose that X is a random variable with E(X) = μ = ̸ 0. Furthermore, supposedly have a suitable function g and we want the distribution of g(X). Applying the Taylor series expansion, we obtain the following:

g(X) ≈ g( μ ) + g ′ ( μ )(X − μ )

If we use g(X) as an estimator for g( μ ), then we can say that we approximately have the following:

E(g(X)) = g( μ )

V(g(X)) = (g ′ ( μ )) 2 V(X)

p Concretely, suppose we want to estimate the odds, 1−p . For example, if p = 2/3, then we say that the odds is 2:1 meaning that the odds of the one outcome are twice 146

3

Statistics

as likely as the odds of the other outcome. Thus, we have g(p) =

p 1−p

and we want

1 to ﬁnd V(g( ˆp)). In our coin-ﬂipping problem, we have the estimator pˆ = n from the Bernoulli-distributed data X k individual coin-ﬂips. Thus,

E( p)ˆ = p

p(1 − p) V( p)ˆ = n

Now, g ′ (p) = 1/(1 − p) 2 , so we have,

V(g( ˆp)) = (g ′ (p)) 2 V( p)ˆ

∑

Xk

1 2 p(1 − p) = ( (1 − p) 2 ) n

p = n(1 − p)3

which is an approximation of the variance of the estimator g( ˆp). Let’s simulate this and see how it agrees.

>>> from scipy import stats

>>> # compute MLE estimates

>>> d=stats.bernoulli(0.1).rvs((10,5000)).mean(0)

>>> # avoid divide-by-zero

>>> d=d[np.logical\_not(np.isclose(d,1))]

>>> # compute odds ratio

>>> odds = d/(1-d)

>>> print ('odds ratio=',np.mean(odds),'var=',np.var(odds))

odds ratio= 0.12289206349206351 var= 0.01797950092214664

The ﬁrst number above is the mean of the simulated odds ratio and the second is the variance of the estimate. According to the variance estimate above, we have V(g(1/10)) ≈ 0.0137, which is not too bad for this approximation. Recall we want to estimate the odds from p.ˆ The code above takes 5000 estimates of the pˆ to estimate V(g). The odds ratio for p = 1/10 is 1/9 ≈ 0.111.

Programming Tip

The code above uses the np.isclose function to identify the ones from the simulation and the np.logical\_not removes these elements from the data because the odds ratio has a zero in the denominator for these values.

Let’s try this again with a probability of heads of 0.5 instead of 0.3. 3.4 Estimation Using Maximum Likelihood

147

Fig. 3.6 The odds ratio is close to linear for small values but becomes unbounded as p approaches one. The delta method is more effective for small underlying values of p, where the linear approximation is better

>>> from scipy import stats

>>> d=stats.bernoulli(.5).rvs((10,5000)).mean(0)

>>> d=d[np.logical\_not(np.isclose(d,1))]

>>> print( 'odds ratio=',np.mean(d),'var=',np.var(d))

odds ratio= 0.499379627776666 var= 0.024512322762879256

The odds ratio in this case is equal to one, which is not close to what was reported. According to our approximation, we should have V(g) = 0.4, which does not look like what our simulation just reported. This is because the approximation is best when the odds ratio is nearly linear and worse otherwise (see Fig.3.6).

3.5 Hypothesis Testing and P-Values

It is sometimes very difﬁcult to unequivocally attribute outcomes to causal factors. For example, did your experiment generate the outcome you were hoping for or not? Maybe something did happen, but the effect is not pronounced enough to separate it from inescapable measurement errors or other factors in the ambient environment? Hypothesis testing is a powerful statistical method to address these questions. Let’s begin by again considering our coin-tossing experiment with unknown parameter p. Recall that the individual coin-ﬂips are Bernoulli distributed. The ﬁrst step is to establish separate hypotheses. First, H 0 is the so-called null hypothesis. In our case this can be

1 H 0 :θ < 2

and the alternative hypothesis is then

1 H 1 :θ ≥ 2 148

3

Statistics

With this setup, the question now boils down to ﬁguring out which hypothesis the data is most consistent with. To choose between these, we need a statistical test that is a function, G, of the sample set X n = { X i } n into the real line, where X i is the heads or tails outcome (X i ∈ { 0, 1 } ). In other words, we compute G(X n ) and check if it exceeds a threshold c. If not, then we declare H 0 (otherwise, declare H 1 ). Notationally, this is the following:

G(X n ) < c ⇒ H0

G(X n ) ≥ c ⇒ H1

In summary, we have the observed data X n and a function G that maps that data onto the real line. Then, using the constant c as a threshold, the inequality effectively divides the real line into two parts, one corresponding to each of the hypotheses.

Whatever this test G is, it will make mistakes of two types—false negatives and false positives. The false positives arise from the case where we declare H 0 when the test says we should declare H 1 . This is summarized in the Table3.1.

For this example, here are the false positives (aka false alarms):

1 P FA = P G(X n ) > c | θ ≤ ( 2 )

Or, equivalently,

P FA = P (G(X n ) > c | H 0 )

Likewise, the other error is a false negative, which we can write analogously as

P F N = P (G(X n ) < c | H 1 )

By choosing some acceptable values for either of these errors, we can solve for the other one. The practice is usually to pick a value of P FA and then ﬁnd the corresponding value of P F N . Note that it is traditional in engineering to speak about detection probability, which is deﬁned as

P D = 1 − P F N = P (G(X n ) > c | H 1 )

In other words, this is the probability of declaring H 1 when the test exceeds the threshold. This is otherwise known as the probability of a true detection or truedetect.

Table 3.1 Truth table for hypotheses testing

Declare H0 H 0 True Correct H 1 True False negative (Type II error)

Declare H1 False positive (Type I error) Correct (true-detect) 3.5 Hypothesis Testing and P-Values

149

3.5.1 Back to the Coin-Flipping Example

In our previous maximum likelihood discussion, we wanted to derive an estimator for the value of the probability of heads for the coin-ﬂipping experiment. For hypothesis testing, we want to ask a softer question: is the probability of heads greater or less than1 /2? As we just established, this leads to the two hypotheses:

1 H 0 :θ < 2

versus,

1 H 1 :θ > 2

Let’s assume we have ﬁve observations. Now we need the G function and a threshold c to help pick between the two hypotheses. Let’s count the number of heads observed in ﬁve observations as our criterion. Thus, we have

5 G(X 5 ) := ∑ Xi i=1

and suppose further that we pick H 1 only if exactly ﬁve out of ﬁve observations are heads. We’ll call this the all-heads test.

Now, because all of the X i are random variables, so is G and we must ﬁnd the corresponding probability mass function for G. Assuming the individual coin tosses are independent, the probability of ﬁve heads is θ 5 . This means that the probability of rejecting the H 0 hypothesis (and choosing H 1 , because there are only two choices here) based on the unknown underlying probability is θ 5 . In the parlance, this is known and the power function as in denoted by β as in

β (θ) = θ5

Let’s get a quick plot this in Fig.3.7.

Now, we have the following false alarm probability:

P FA = P(G(X n ) = 5 | H 0 ) = P(θ 5 | H 0 )

Notice that this is a function of θ, which means there are many false alarm probability values that correspond to this test. To be on the conservative side, we’ll pick the supremum (i.e., maximum) of this function, which is known as the size of the test, traditionally denoted by α,

α = sup β (θ) θ∈Θ 0 150

3

Statistics

Fig. 3.7 Power function for the all-heads test. The dark circle indicates the value of the function indicating α

with domain Θ 0 = { θ < 1/2 } which in our case is

1 5 α = sup θ 5 = = 0.03125 ( 2 ) θ< 1 2

Likewise, for the detection probability,

P D (θ) = P(θ 5 | H 1 )

which is again a function of the parameter θ. The problem with this test is that the P D is pretty low for most of the domain of θ. For instance, values in the nineties for P D only happen when θ > 0.98. In other words, if the coin produces heads 98 times out of 100, then we can detect H 1 reliably. Ideally, we want a test that is zero for the domain corresponding to H 0 (i.e., Θ 0 ) and equal to one otherwise. Unfortunately, even if we increase the length of the observed sequence, we cannot escape this effect with this test. You can try plotting θ n for larger and larger values of n to see this.

Majority Vote Test. Due to the problems with the detection probability in the allheads test, maybe we can think of another test that will have the performance we want? Suppose we reject H 0 if the majority of the observations are heads. Then, using the same reasoning as above, we have

5 5 β (θ) = ∑ (1 − θ)5−k θk k ) k=3 (

Figure3.8 shows the power function for both the majority vote and the all-heads tests. 3.5 Hypothesis Testing and P-Values

151

Fig. 3.8 Compares the power function for the all-heads test with that of the majority vote test

In this case, the new test has size

1 α = sup θ 5 + 5θ 4 (−θ + 1) + 10θ 3 (−θ + 1) 2 = 2 θ< 1 2

As before we only get to upward of 90% for detection probability only when the underlying parameter θ > 0.75. Let’s see what happens when we consider more than ﬁve samples. For example, let’s suppose that we have n = 100 samples and we want to vary the threshold for the majority vote test. For example, let’s have a new test where we declare H 1 when k = 60 out of the 100 trials turns out to be heads. What is the β function in this case?

100 100 β (θ) = ∑ (1 − θ)100−k θk k ) k=60 (

This is too complicated to write by hand, but the statistics module in Sympy has all the tools we need to compute this.

>>> from sympy.stats import P, Binomial

>>> theta = S.symbols('theta',real=True)

>>> X = Binomial('x',100,theta)

>>> beta\_function = P(X>60)

>>> print (beta\_function.subs(theta,0.5)) # alpha

0.0176001001088524

>>> print (beta\_function.subs(theta,0.70))

0.979011423996075

These results are much better than before because the β function is much steeper. If we declare H 1 when we observe 60 out of 100 trials are heads, then we wrongly declare heads approximately 1.8% of the time. Otherwise, if it happens that the true value for p > 0.7, we will conclude correctly approximately 97% of the time. A quick simulation can sanity check these results as shown below: 152

3

Statistics

>>> from scipy import stats

>>> rv=stats.bernoulli(0.5) # true p = 0.5

>>> # number of false alarms ˜0.018

>>> print (sum(rv.rvs((1000,100)).sum(axis=1)>60)/1000.)

0.025

The above code is pretty dense so let’s unpack it. In the ﬁrst line, we use the scipy.stats module to deﬁne the Bernoulli random variable for the coin-ﬂip. Then, we use the rvs method of the variable to generate 1000 trials of the experiment where each trial consists of 100 coin-ﬂips. This generates a 1000 × 100 matrix where the rows are the individual trials and the columns are the outcomes of each respective set of 100 coin-ﬂips. The sum(axis=1) part computes the sum across the columns. Because the values of the embedded matrix are only 1 or 0 this gives us the count of ﬂips that are heads per row. The next >60 part computes the boolean 1000-long vector of values that are bigger than 60. The ﬁnal sum adds these up. Again, because the entries in the array are True or False the sum computes the count of times the number of heads has exceeded 60 per 100 coin-ﬂips in each of 1000 trials. Then, dividing this number by 1000 gives a quick approximation of false alarm probability we computed above for this case where the true value of p = 0.5.

3.5.2 Receiver Operating Characteristic

Because the majority vote test is a binary test, we can compute the receiver operating characteristic (ROC) which is the graph of the (P FA , P D ). The term comes from radar systems but is a very general method for consolidating all of these issues into a single graph. Let’s consider a typical signal processing example with two hypotheses. In H 0 , there is noise but no signal present at the receiver,

H 0 : X =

where ∼ N(0, σ 2 ) represents additive noise. In the alternative hypothesis, there is a deterministic signal at the receiver,

H 1 : X = μ +

Again, the problem is to choose between these two hypotheses. For H 0 , we have X ∼ N(0, σ 2 ) and for H 1 , we have X ∼ N( μ , σ 2 ). Recall that we only observe values for x and must pick either H 0 or H 1 from these observations. Thus, we need a threshold, c, to compare x against in order to distinguish the two hypotheses. Figure3.9 shows the probability density functions under each of the hypotheses. The dark vertical line is the threshold c. The gray shaded area is the probability of detection, P D and the shaded area is the probability of false alarm, P FA . The test evaluates every observation of x and concludes H 0 if x < c and H 1 otherwise. 3.5 Hypothesis Testing and P-Values

153

Fig. 3.9 The two density functions for the H 0 and H 1 hypotheses. The shaded gray area is the detection probability and the shaded dark gray area is the probability of false alarm. The vertical line is the decision threshold

Programming Tip

The shading shown in Fig.3.9 comes from Matplotlib’s fill\_between function. This function has a where keyword argument to specify which part of the plot to apply shading with speciﬁed color keyword argument. Note there is also a fill\_betweenx function that ﬁlls horizontally. The text function can A place formatted text anywhere in the plot and can utilize basic L T E X formatting.

As we slide the threshold left and right along the horizontal axis, we naturally changethecorrespondingareasundereachofthecurvesshowninFig.3.9andthereby change the values of P D and P FA . The contour that emerges from sweeping the threshold this way is the ROC as shown in Fig.3.10. This ﬁgure also shows the diagonal line which corresponds to making decisions based on the ﬂip of a fair coin. Any meaningful test must do better than coin-ﬂipping so the more the ROC bows up to the top left corner of the graph, the better. Sometimes ROCs are quantiﬁed into a single number called the area under the curve (AUC), which varies from 0.5 to 1.0 as shown. In our example, what separates the two probability density functions is the value of μ . In a real situation, this would be determined by signal processing methods that include many complicated trade-offs. The key idea is that whatever those trade-offs are, the test itself boils down to the separation between these two density functions—good tests separate the two density functions and bad tests do not. Indeed, when there is no separation, we arrive at the diagonal-line coin-ﬂipping situation we just discussed.

What values for P D and P FA are considered acceptable depends on the application. For example, suppose you are testing for a fatal disease. It could be that you are willing to except a relatively high P FA value if that corresponds to a good 154

3

Statistics

Fig. 3.10 The receiver operating characteristic (ROC) corresponding to Fig.3.9

P D because the test is relatively cheap to administer compared to the alternative of missing a detection. On the other hand, may be a false alarm triggers an expensive response, so that minimizing these alarms is more important than potentially missing a detection. These trade-offs can only be determined by the application and design factors.

3.5.3 P-Values

There are a lot of moving parts in hypothesis testing. What we need is a way to consolidate the ﬁndings. The idea is that we want to ﬁnd the minimum level at which the test rejects H 0 . Thus, the p-value is the probability, under H 0 , that the test statistic is at least as extreme as what was actually observed. Informally, this means that smaller values imply that H 0 should be rejected, although this doesn’t mean that large values imply that H 0 should be retained. This is because a large p-value can arise from either H 0 being true or the test having low statistical power.

If H 0 is true, the p-value is uniformly distributed in the interval (0, 1). If H 1 is true, the distribution of the p-value will concentrate closer to zero. For continuous distributions, this can be proven rigorously and implies that if we reject H 0 when the corresponding p-value is less than α, then the probability of a false alarm is α. Perhaps it helps to formalize this a bit before computing it. Suppose τ(X) is a test statistic that rejects H 0 as it gets bigger. Then, for each sample x, corresponding to the data we actually have on-hand, we deﬁne

p(x) = sup P θ (τ(X) > τ(x)) θ∈Θ 0 3.5 Hypothesis Testing and P-Values

155

This equation states that the supremum (i.e., maximum) probability that the test statistic, τ(X), exceeds the value for the test statistic on this particular data (τ(x)) over the domain Θ 0 is deﬁned as the p-value. Thus, this embodies a worst-case scenario over all values of θ.

Here’s one way to think about this. Suppose you rejected H 0 , and someone says that you just got lucky and somehow just drew data that happened to correspond to a rejection of H 0 . What p-values provide is a way to address this by capturing the odds of just a favorable data-draw. Thus, suppose that your p-value is 0.05. Then, what you are showing is that the odds of just drawing that data sample, given H 0 is in force, is just 5%. This means that there’s a 5% chance that you somehow lucked out and got a favorable draw of data.

Let’s make this concrete with an example. Given, the majority vote rule above, suppose we actually do observe three of ﬁve heads. Given the H 0 , the probability of observing this event is the following:

5 5 1 p(x) = sup ∑ ( k ) θ k (1 − θ) 5−k = 2 θ∈Θ 0 k=3

For the all-heads test, the corresponding computation is the following:

1 p(x) = sup θ 5 = = 0.03125 θ∈Θ 0 25

From just looking at these p-values, you might get the feeling that the second test is better, but we still have the same detection probability issues we discussed above; so, p-values help in summarizing some aspects of our hypothesis testing, but they do not summarize all the salient aspects of the entire situation.

3.5.4 Test Statistics

As we have seen, it is difﬁcult to derive good test statistics for hypothesis testing without a systematic process. The Neyman–Pearson Test is derived from ﬁxing a false alarm value (α) and then maximizing the detection probability. This results in the Neyman–Pearson Test,

fX (x) H 1 H L(x) = | 1 ≷ γ f X | H 0 (x) H 0

where L is the likelihood ratio and where the threshold γ is chosen such that

f X | H 0 (x)dx = α ∫ x:L(x)> γ 156

3

Statistics

The Neyman–Pearson Test is one of a family of tests that use the likelihood ratio.

Example. Suppose we have a receiver and we want to distinguish whether just noise (H 0 ) or signal pulse noise (H 1 ) is received. For the noise-only case, we have x ∼ N(0, 1) and for the signal pulse noise case we have x ∼ N(1, 1). In other words, the mean of the distribution shifts in the presence of the signal. This is a very common problem in signal processing and communications. The Neyman–Pearson Test then boils down to the following:

H 1 L(x) = e − 2 1 + x ≷ γ

H 0

Now we have to ﬁnd the threshold γ that solves the maximization problem that characterizes the Neyman–Pearson Test. Taking the natural logarithm and re-arranging gives

H 1 1 x ≷ log + γ H 0 2

The next step is ﬁnd γ corresponding to the desired α by computing it from the following:

∞ f X | H 0 (x)dx = α ∫ 1/2 + log γ

For example, taking α = 1/100, gives γ we have,

≈ 6.21. To summarize the test in this case,

H 1 H 0

x ≷ 2.32

Thus, if we measure X and see that its value exceeds the threshold above, we declare H 1 and otherwise declare H 0 . The following code shows how to solve this example using Sympy and Scipy. First, we set up the likelihood ratio,

>>> import sympy as S

>>> from sympy import stats

>>> s = stats.Normal('s',1,1) # signal+noise

>>> n = stats.Normal('n',0,1) # noise

>>> x = S.symbols('x',real=True)

>>> L = stats.density(s)(x)/stats.density(n)(x)

Next, to ﬁnd the γ value,

>>> g = S.symbols('g',positive=True) # define gamma

>>> v=S.integrate(stats.density(n)(x),

...

(x,S.Rational(1,2)+S.log(g),S.oo)) 3.5 Hypothesis Testing and P-Values

157

Programming Tip

Providing additional information regarding the Sympy variable by using the keywordargument positive=Truehelpstheinternalsimpliﬁcationalgorithms work faster and better. This is especially useful when dealing with complicated integrals that involve special functions. Furthermore, note that we used the Rational function to deﬁne the 1/2 fraction, which is another way of providing hints to Sympy. Otherwise, it’s possible that the ﬂoating-point representation of the fraction could disguise the simple fraction and thereby miss internal simpliﬁcation opportunities.

We want to solve for g in the above expression. Sympy has some built-in numerical solvers as in the following:

>>> print (S.nsolve(v-0.01,3.0)) # approx 6.21 6.21116124253284

Note that in this situation it is better to use the numerical solvers because Sympy solve may grind along for a long time to resolve this.

Generalized Likelihood Ratio Test. The likelihood ratio test can be generalized using the following statistic:

ˆ supθ∈Θ 0 L(θ) L( θ 0 ) Λ(x) = = ˆ supθ∈Θ L(θ) L( θ)

ˆ where θ 0 ˆ maximizes L(θ) subject to θ ∈ Θ 0 and θ is the maximum likelihood estimator. The intuition behind this generalization of the Likelihood Ratio Test is that the denominator is the usual maximum likelihood estimator and the numerator is the maximum likelihood estimator, but over a restricted domain (Θ 0 ). This means that the ratio is always less than unity because the maximum likelihood estimator over the entire space will always be at least as maximal as that over the more restricted space. When this Λ ratio gets small enough, it means that the maximum likelihood estimator over the entire domain (Θ) is larger which means that it is safe to reject the null hypothesis H 0 . The tricky part is that the statistical distribution of Λ is usually eye-wateringly difﬁcult. Fortunately, Wilks Theorem says that with sufﬁciently large n, the distribution of −2 log Λ is approximately chi-square with r − r 0 degrees of freedom, where r is the number of free parameters for Θ and r 0 is the number of free parameters in Θ 0 . With this result, if we want an approximate test at level α, we can reject H 0 when −2 log Λ ≥ χ r−r 0 2 (α) where χ r−r 0 2 (α) denotes the 1 − α quantile of the χ r−r 0 2 chi-square distribution. However, the problem with this result is that there is no deﬁnite way of knowing how big n should be. The advantage of this generalized likelihood ratio test is that it can test multiple hypotheses simultaneously, as illustrated in the following example. 158

3

Statistics

Example. Let’s return to our coin-ﬂipping example, except now we have three different coins. The likelihood function is then,

L(p 1 , p 2 , p 3 ) = binom(k 1 ; n 1 , p 1 )binom(k 2 ; n 2 , p 2 )binom(k 3 ; n 3 , p 3 )

where binom is the binomial distribution with the given parameters. For example,

n n binom(k ; n, p) = ∑ k p k (1 − p)n−k ) k=0 (

Thenullhypothesisisthatallthreecoinshavethesameprobabilityofheads, H 0 : p = p 1 = p 2 = p 3 . The alternative hypothesis is that at least one of these probabilities is different. Let’s consider the numerator of the Λ ﬁrst, which will give us the maximum likelihood estimator of p. Because the null hypothesis is that all the p values are equal, we can just treat this as one big binomial distribution with n = n 1 + n 2 + n3 and k = k 1 + k 2 + k 3 is the total number of heads observed for any coin. Thus, under the null hypothesis, the distribution of k is binomial with parameters n and p. Now, what is the maximum likelihood estimator for this distribution? We have worked this problem before and have the following:

k p0 ˆ = n

In other words, the maximum likelihood estimator under the null hypothesis is the proportion of ones observed in the sequence of n trials total. Now, we have to substitute this in for the likelihood under the null hypothesis to ﬁnish the numerator of Λ,

L( p0 ˆ , p0 ˆ , p0 ˆ ) = binom(k 1 ; n 1 , p0 ˆ )binom(k 2 ; n 2 , p0 ˆ )binom(k 3 ; n 3 , p0 ˆ )

For the denominator of Λ, which represents the case of maximizing over the entire space, the maximum likelihood estimator for each separate binomial distribution is likewise,

ki pi ˆ = ni

which makes the likelihood in the denominator the following:

L( p1 ˆ , p2 ˆ , p3 ˆ ) = binom(k 1 ; n 1 , p1 ˆ )binom(k 2 ; n 2 , p2 ˆ )binom(k 3 ; n 3 , p3 ˆ )

for each of the i ∈ { 1, 2, 3 } binomial distributions. Then, the Λ statistic is then the following:

L( p0 , p0 ˆ , p0 ˆ ) Λ(k 1 , k 2 , k 3 ) = L( p1 ˆ , p2 ˆ , p3 ˆ ) 3.5 Hypothesis Testing and P-Values

159

Wilks theorems state that −2 log Λ is chi-square distributed. We can compute this example with the statistics tools in Sympy and Scipy.

>>> from scipy.stats import binom, chi2

>>> import numpy as np

>>> # some sample parameters

>>> p0,p1,p2 = 0.3,0.4,0.5

>>> n0,n1,n2 = 50,180,200

>>> brvs= [ binom(i,j) for i,j in zip((n0,n1,n2),(p0,p1,p2))]

>>> def gen\_sample(n=1):

... ... ... ... ... ...

'generate samples from separate binomial distributions'

if

n==1:

return [i.rvs() for i in brvs] else:

return [gen\_sample() for k in range(n)]

Programming Tip

Note the recursion in the deﬁnition of the gen\_sample function where a conditional clause of the function calls itself. This is a quick way to reusing code and generating vectorized output. Using np.vectorize is another way, but the code is simple enough in this case to use the conditional clause. In Python, it is generally bad for performance to have code with nested recursion because of how the stack frames are managed. However, here we are only recursing once so this is not an issue.

Next, we compute the logarithm of the numerator of the Λ statistic,

>>> k0,k1,k2 = gen\_sample()

>>> print (k0,k1,k2) 12 68 103

>>> pH0 = sum((k0,k1,k2))/sum((n0,n1,n2))

>>> numer = np.sum([np.log(binom(ni,pH0).pmf(ki)) ... for ni,ki in ... zip((n0,n1,n2),(k0,k1,k2))])

>>> print (numer)

-15.545863836567879

Note that we used the null hypothesis estimate for the p0 ˆ . Likewise, for the logarithm of the denominator we have the following:

>>> denom = np.sum([np.log(binom(ni,pi).pmf(ki)) ... for ni,ki,pi in ... zip((n0,n1,n2),(k0,k1,k2),(p0,p1,p2))]) 160

3

Statistics

>>> print (denom) -8.424106480792402

Now, we can compute the logarithm of the Λ statistic as follows and see what the corresponding value is according to Wilks theorem,

>>> chsq=chi2(2)

>>> logLambda =-2\*(numer-denom)

>>> print (logLambda)

14.243514711550954

>>> print (1- chsq.cdf(logLambda))

0.0008073467083287156

Because the value reported above is less than the 5% signiﬁcance level, we reject the null hypothesis that all the coins have the same probability of heads. Note that there are two degrees of freedom because the difference in the number of parameters between the null hypothesis (p) and the alternative (p 1 , p 2 , p 3 ) is two. We can build a quick Monte Carlo simulation to check the probability of detection for this example using the following code, which is just a combination of the last few code blocks,

>>> c= chsq.isf(.05) # 5% significance level

>>> out = []

>>> for k0,k1,k2 in gen\_sample(100):

... pH0 = sum((k0,k1,k2))/sum((n0,n1,n2)) ... numer = np.sum([np.log(binom(ni,pH0).pmf(ki)) ... for ni,ki in ... zip((n0,n1,n2),(k0,k1,k2))]) ... denom = np.sum([np.log(binom(ni,pi).pmf(ki)) ... for ni,ki,pi in ... zip((n0,n1,n2),(k0,k1,k2),(p0,p1,p2))]) ... out.append(-2\*(numer-denom)>c) ...

>>> print (np.mean(out)) # estimated probability of detection

0.59

The above simulation shows the estimated probability of detection, for this set of example parameters. This relative low probability of detection means that while the test is unlikely (i.e., at the 5% signiﬁcance level) to mistakenly pick the null hypothesis, it is likewise missing many of the H 1 cases (i.e., low probability of detection). The trade-off between which is more important is up to the particular context of the problem. In some situations, we may prefer additional false alarms in exchange for missing fewer H 1 cases.

Permutation Test. The Permutation Test is good way to test whether or not samples come from the same distribution. For example, suppose that

X1 , X2 ,..., X m ∼ F 3.5 Hypothesis Testing and P-Values

161

and also,

Y1 ,Y2 ,...,Y n ∼ G

That is, Y i and X i come from different distributions. Suppose we have some test statistic, for example

T (X 1 , . . . , X m , Y 1 , . . . , Y n ) = | X − Y |

Under the null hypothesis for which F = G, any of the (n + m)! permutations are equally likely. Thus, suppose for each of the (n + m)! permutations, we have the computed statistic,

{ T 1 , T 2 , . . . , T (n + m)! }

Then, under thenull hypothesis, eachof thesevalues is equallylikely. Thedistribution of T under the null hypothesis is the permutation distribution that puts weight 1/(n + m)! on each T -value. Suppose t o is the observed value of the test statistic and assume that large T rejects the null hypothesis, then the p-value for the permutation test is the following:

(n + m)! 1 P(T > t o ) = ∑ I (T j > t o ) (n + m)! j=1

where I () is the indicator function. For large (n + m)!, we can sample randomly from the set of all permutations to estimate this p-value.

Example. Let’s return to our coin-ﬂipping example from last time, but now we have only two coins. The hypothesis is that both coins have the same probability of heads. We can use the built-in function in Numpy to compute the random permutations.

>>> x=binom(10,0.3).rvs(5) # p=0.3

>>> y=binom(10,0.5).rvs(3) # p=0.5

>>> z = np.hstack([x,y]) # combine into one array

>>> t\_o = abs(x.mean()-y.mean())

>>> out = [] # output container

>>> for k in range(1000):

... perm = np.random.permutation(z) ... T=abs(perm[:len(x)].mean()-perm[len(x):].mean()) ... out.append((T>t\_o)) ...

>>> print ('p-value = ', np.mean(out)) p-value = 0.0

Note that the size of total permutation space is 8! = 40320 so we are taking relatively few (i.e., 100) random permutations from this space.

Wald Test. The Wald Test is an asymptotic test. Suppose we have H 0 : θ = θ 0 and otherwise H 1 : θ = ̸ θ 0 , the corresponding statistic is deﬁned as the following: 162

3

Statistics

θ n ˆ − θ0 W = se

where

ˆ θ is the maximum likelihood estimator and se is the standard error,

ˆ se = √ V( θ n )

d Under general conditions, W → N(0, 1). Thus, an asymptotic test at level α rejects when | W | > z α/2 where z α/2 corresponds to P( | Z | > z α/2 ) = α with Z ∼ N(0, 1). For our favorite coin-ﬂipping example, if H 0 : θ = θ 0 , then

θ ˆ − θ0 W = √ θ(1 − θ)/n

We can simulate this using the following code at the usual 5% signiﬁcance level,

>>> from scipy import stats

>>> theta0 = 0.5 # H0

>>> k=np.random.binomial(1000,0.3)

>>> theta\_hat = k/1000. # MLE

>>> W = (theta\_hat-theta0)/np.sqrt(theta\_hat\*(1-theta\_hat)/1000)

>>> c = stats.norm().isf(0.05/2) # z\_{alpha/2}

>>> print (abs(W)>c) # if true, reject H0 True

This rejects H 0 because the true θ = 0.3 and the null hypothesis is that θ = 0.5. Note that n = 1000 in this case which puts us well inside the asymptotic range of the result. We can re-do this example to estimate the detection probability for this example as in the following code:

>>> theta0 = 0.5 # H0

>>> c = stats.norm().isf(0.05/2.) # z\_{alpha/2}

>>> out = []

>>> for i in range(100):

... k=np.random.binomial(1000,0.3) ... theta\_hat = k/1000. # MLE ... W = (theta\_hat-theta0)/np.sqrt(theta\_hat\*(1-theta\_hat)/1000.) ... out.append(abs(W)>c) # if true, reject H0 ...

>>> print (np.mean(out)) # detection probability

1.0 3.5 Hypothesis Testing and P-Values

163

3.5.5 Testing Multiple Hypotheses

Thus far, we have focused primarily on two competing hypotheses. Now, we consider multiple comparisons. The general situation is the following. We test the null hypothesis against a sequence of n competing hypotheses H k . We obtain p-values for each hypothesis so now we have multiple p-values to consider { p k } . To boil this sequence down to a single criterion, we can make the following argument. Given n independent hypotheses that are all untrue, the probability of getting at least one false alarm is the following:

P FA = 1 − (1 − p 0 )n

where p 0 is the individual p-value threshold (say, 0.05). The problem here is that P FA → 1 as n → ∞. If we want to make many comparisons at once and control the overall false alarm rate the overall p-value should be computed under the assumption that none of the competing hypotheses is valid. The most common way to address this is with the Bonferroni correction which says that the individual signiﬁcance level should be reduced to p/n. Obviously, this makes it much harder to declare significance for any particular hypothesis. The natural consequence of this conservative restriction is to reduce the statistical power of the experiment, thus making it more likely the true effects will be missed.

In 1995, Benjamini and Hochberg devised a simple method that tells which p-values are statistically signiﬁcant. The procedure is to sort the list of p-values in ascending order, choose a false-discovery rate (say, q), and then ﬁnd the largest p-value in the sorted list such that p k ≤ kq/n, where k is the p-value’s position in the sorted list. Finally, declare that p k value and all the others less than it statistically signiﬁcant. This procedure guarantees that the proportion of false positives is less than q (on average). The Benjamini–Hochberg procedure (and its derivatives) is fast and effective and is widely used for testing hundreds of primarily false hypotheses when studying genetics or diseases. Additionally, this procedure provides better statistical power than the Bonferroni correction.

3.5.6 Fisher Exact Test

Contingency tables represent the partitioning of a sample population of two categories between two different classiﬁcations as shown in the following Table3.2. The

Table 3.2 Example contingency table Infection Male 13 Female 12 Total 25

No infection 11 1 12

Total 24 13 37 164

3

Statistics

question is whether or not the observed table corresponds to a random partition of the sample population, constrained by the marginal sums. Note that because this is a two-by-two table, a change in any of the table entries automatically affects all of the other terms because of the row and column sum constraints. This means that equivalent questions like “Under a random partition, what is the probability that a particular table entry is at least as large as a given value?” can be meaningfully posed.

The Fisher Exact Test addresses this question. The idea is to compute the probability of a particular entry of the table, conditioned upon the marginal row and column sums,

P(X i, j | r 1 ,r 2 , c 1 , c 2 )

where X i, j is (i, j) table entry, r 1 represents the sum of the ﬁrst row, r 2 represents the sum of the second row, c 1 represents the sum of the ﬁrst column, and c 2 is the sum of the second column. This probability is given by the hypergeometric distribution. Recall that the hypergeometric distribution gives the probability of sampling (without replacement) k items from a population of N items consisting of exactly two different kinds of items,

K N−K P(X = k) = ( k )( N n−k ) (n )

where N is the population size, K is the total number of possible favorable draws, n is the number of draws, and k is the number of observed favorable draws. With the corresponding identiﬁcation of variables, the hypergeometric distribution gives the desired conditional probability: K = r 1 , k = x, n = c 1 , N = r 1 + r 2 .

In the example of the Table3.2, the probability for x = 13 male infections among a population of r 1 = 24 males in a total population of c 1 = 25 infected persons, including r 2 = 13 females. The scipy.stats module has the Fisher Exact Test implemented as shown below:

>>> import scipy.stats

>>> table = [[13,11],[12,1]]

>>> odds\_ratio, p\_value=scipy.stats.fisher\_exact(table)

>>> print(p\_value)

0.02718387758955712

The default for scipy.stats.fisher\_exact is the two-sided test. The following result is for the less option,

>>> import scipy.stats

>>> odds\_ratio, p\_value=scipy.stats.fisher\_exact(table,alternative='less')

>>> print(p\_value)

0.018976707519532877

This means that the p-value is computed by summing over the probabilities of contingency tables that are less extreme than the given table. To understand what this means, 3.5 Hypothesis Testing and P-Values

165

we can use the scipy.stats.hypergeom function to compute the probabilities of these with the number of infected men is less than or equal to 13.

>>> hg = scipy.stats.hypergeom(37, 24, 25)

>>> probs = [(hg.pmf(i)) for i in range(14)]

>>> print (probs)

[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,

0.0014597467322717626, 0.017516960787261115]

>>> print(sum(probs))

0.018976707519532877

This is the same as the prior p-value result we obtained from scipy.stats. fisher\_exact. Another option is greater which derives from the following analogous summation:

>>> odds\_ratio, p\_value=scipy.stats.fisher\_exact(table,alternative='greater')

>>> probs = [hg.pmf(i) for i in range(13,25)]

>>> print(probs)

[0.017516960787261115, 0.08257995799708828, 0.2018621195484381,

0.28386860561499044, 0.24045340710916852, 0.12467954442697629,

0.039372487713781906, 0.00738234144633414, 0.0007812001530512284,

4.261091743915799e-05, 1.0105355914424832e-06, 7.017608273906114e-09]

>>> print(p\_value)

0.9985402532677288

>>> print(sum(probs))

0.9985402532677288

Finally, the two-sided version excludes those individual table probabilities that are less that of the given table

>>> \_,p\_value=scipy.stats.fisher\_exact(table)

>>> probs = [ hg.pmf(i) for i in range(25) ]

>>> print(sum(i for i in probs if i<= hg.pmf(13)))

0.027183877589557117

>>> print(p\_value)

0.02718387758955712

Thus, for this particular contingency table, we could reasonably conclude that 13 infected males in this total population is statistically signiﬁcant with a p-value less than ﬁve percent.

Performing this kind of analysis for tables larger than 2x2 easily becomes computationally challenging due to the nature of the underlying combinatorics and usually requires specialized approximations.

In this section, we discussed the structure of statistical hypothesis testing and deﬁned the various terms that are commonly used for this process, along with the illustrations of what they mean in our running coin-ﬂipping example. From an engineering standpoint, hypothesis testing is not as common as conﬁdence intervals and point estimates. On the other hand, hypothesis testing is very common in social and medical science, where one must deal with practical constraints that may limit the 166

3

Statistics

sample size or other aspects of the hypothesis testing rubric. In engineering, we can usually have much more control over the samples and models we employ because they are typically inanimate objects that can be measured repeatedly and consistently. This is obviously not so with human studies, which generally have other ethical and legal considerations.

3.6 Conﬁdence Intervals

In a previous coin-ﬂipping discussion, we discussed estimation of the underlying probability of getting a heads. There, we derived the estimator as

n 1 pn ˆ = ∑ Xi n i=1

where X i ∈ { 0, 1 } . Conﬁdence intervals allow us to estimate how close we can get to the true value that we are estimating. Logically, that seems strange, doesn’t it? We really don’t know the exact value of what we are estimating (otherwise, why estimate it?), and yet, somehow we know how close we can get to something we admit we don’t know? Ultimately, we want to make statements like the probability of the value in a certain interval is 90%. Unfortunately, that is something we will not be able to say using our methods. Note that Bayesian estimation gets closer to this statement by using credible intervals, but that is a story for another day. In our situation, the best we can do is say roughly the following: if we ran the experiment multiple times, then the conﬁdence interval would trap the true parameter 90% of the time.

Let’s return to our coin-ﬂipping example and see this in action. One way to get at a conﬁdence interval is to use Hoeffding’s inequality from Sect.2.11.3 specialized to our Bernoulli variables as

P( | pn ˆ − p | > ) ≤ 2 exp(−2n 2 )

Now, we can form the interval I = [ pn ˆ − n , pn ˆ constructed as

1 2 log n = √ 2n α

+ n

]

, where n is carefully

which makes the right side of the Hoeffding inequality equal to α. Thus, we ﬁnally have

P(p ∈ / I) = P ( | pn ˆ − p | > n ) ≤ α

Thus, P(p ∈ I) ≥ 1 − α. As a numerical example, let’s take n = 100, α = 0.05, then plugging into everything we have given n = 0.136. So, the 95% conﬁdence interval here is therefore 3.6 Conﬁdence Intervals

167

I = [ pn ˆ − n , pn ˆ + n ] = [ pn ˆ − 0.136, pn ˆ + 0.136 ]

The following code sample is a simulation to see if we can really trap the underlying parameter in our conﬁdence interval.

>>> from scipy import stats

>>> import numpy as np

>>> b= stats.bernoulli(.5) # fair coin distribution

>>> nsamples = 100

>>> # flip it nsamples times for 200 estimates

>>> xs = b.rvs(nsamples\*200).reshape(nsamples,-1)

>>> phat = np.mean(xs,axis=0) # estimated p

>>> # edge of 95% confidence interval

>>> epsilon\_n=np.sqrt(np.log(2/0.05)/2/nsamples) ... 0.5 <= (epsilon\_n +phat) ... ).mean()\*100

>>> pct=np.logical\_and(phat-epsilon\_n<=0.5,

>>> print ('Interval trapped correct value ', pct,'% of the time') Interval trapped correct value 99.5 % of the time

The result shows that the estimator and the corresponding interval was able to trap the true value at least 95% of the time. This is how to interpret the action of conﬁdence intervals.

However, the usual practice is to not use Hoeffding’s inequality and instead use arguments around asymptotic normality. The deﬁnition of the standard error is the following:

ˆ se = √ V( θ n )

ˆ θn

where is the point estimator for the parameter θ, given n samples of data X n , and ˆ ˆ V( θ n ) is the variance of θ n . Likewise, the estimated standard error is se.⌃ For example, in our coin-ﬂipping example, the estimator was pˆ = ∑ X i /n with corresponding variance V( pn ˆ ) = p(1− p)/n. Plugging in the point estimate gives us the estimated standard error: se⌃ = √ ˆp(1 − ˆp)/n. Because maximum likelihood estimators are asymptotically normal, 4 we know that pn ˆ ∼ N(p, se⌃ 2 ). Thus, if we want a 1 − α conﬁdence interval, we can compute

P( | pn ˆ − p | < ξ ) > 1 − α

but since we know that ( pn ˆ − p) is asymptotically normal, N(0, se⌃ 2 ), we can instead compute

ξ N(0, se⌃ 2 )dx > 1 − α ∫ − ξ

4 Certaintechnicalregularityconditionsmustholdforthispropertyofmaximumlikelihoodestimator to work. See [2] for more details. 168

3

Statistics

Fig. 3.11 The gray circles are the point estimates that are bounded above and below by both asymptotic conﬁdence intervals and Hoeffding intervals. The asymptotic intervals are tighter because the underpinning asymptotic assumptions are valid for these estimates

This looks ugly to compute because we need to ﬁnd ξ , but Scipy has everything we need for this.

>>> # compute estimated se for all trials

>>> se=np.sqrt(phat\*(1-phat)/xs.shape[0])

>>> # generate random variable for trial 0

>>> rv=stats.norm(0, se[0])

>>> # compute 95% confidence interval for that trial 0

>>> np.array(rv.interval(0.95))+phat[0] array([0.42208023, 0.61791977])

>>> def compute\_CI(i):

...

return

stats.norm.interval(0.95,loc=i,

... scale=np.sqrt(i\*(1-i)/xs.shape[0])) ...

>>> lower,upper = compute\_CI(phat)

Figure3.11 shows the asymptotic conﬁdence intervals and the Hoeffding-derived conﬁdence intervals. As shown, the Hoeffding intervals are a bit more generous than the asymptotic estimates. However, this is only true so long as the asymptotic approximation is valid. In other words, there exists some number of n samples for which the asymptotic intervals may not work. So, even though they may be a bit more generous, the Hoeffding intervals do not require arguments about asymptotic convergence. In practice, nonetheless, asymptotic convergence is always in play (even if not explicitly stated).

Conﬁdence Intervals and Hypothesis Testing. It turns out that there is a close dual relationship between hypothesis testing and conﬁdence intervals. To see this in action, consider the following hypothesis test for a normal distribution, H 0 : μ = μ0 versus H 1 : μ = ̸ μ 0 . A reasonable test has the following rejection region:

σ x : | ¯x − μ 0 | > zα/2 { √ n } 3.6 Conﬁdence Intervals

169

where P(Z > z α/2 ) = α/2 and P(−z α/2 < Z < z α/2 ) = 1 − α and where

Z ∼ N(0, 1). This is the same thing as saying that the region corresponding to acceptance of H 0 is then,

σ σ ¯x − z α/2 ≤ ¯x + zα/2 ≤ μ0 √ n √ n

(3.6.0.1)

Because the test has size α, the false alarm probability, P(H 0 rejected | μ = μ 0 ) = α. Likewise, the P(H 0 accepted | μ = μ 0 ) = 1 − α. Putting this all together with interval deﬁned above means that

σ σ P ¯x − z α/2 ≤ ¯x + z α/2 = 1 − α H0 ≤ μ0 ( √ n √ n )

Because this is valid for any μ 0 , we can drop the H 0 condition and say the following:

σ σ P ¯x − z α/2 ≤ ¯x + zα/2 ≤ μ 0 =1− α ( √ n √ n )

As may be obvious by now, the interval in Eq.3.6.0.1 above is the 1 − α conﬁdence interval! Thus, we have just obtained the conﬁdence interval by inverting the acceptance region of the level α test. The hypothesis test ﬁxes the parameter and then asks what sample values (i.e., the acceptance region) are consistent with that ﬁxed value. Alternatively, the conﬁdence interval ﬁxes the sample value and then asks what parameter values (i.e., the conﬁdence interval) make this sample value most plausible. Note that sometimes this inversion method results in disjoint intervals (known as conﬁdence sets).

3.7 Linear Regression

Linear regression gets to the heart of statistics: Given a set of data points, what is the relationship of the data in hand to data yet seen? How should information from one dataset propagate to other data? Linear regression offers the following model to address this question:

E(Y | X = x) ≈ ax + b

That is, given speciﬁc values for X, assume that the conditional expectation is a linear function of those speciﬁc values. However, because the observed values are not the expectations themselves, the model accommodates this with an additive noise term. In other words, the observed variable (a.k.a. response, target, dependent variable) is modeled as

E(Y | X = x i ) + i ≈ ax + b + i = y 170

3

Statistics

where E( i ) = 0 and the i are iid and where the distribution function of i depends on the problem, even though it is often assumed Gaussian. The X = x values are known as independent variables, covariates, or regressors.

Let’s see if we can use all of the methods we have developed so far to understand this form of regression. The ﬁrst task is to determine how to estimate the unknown linear parameters, a and b. To make this concrete, let’s assume that ∼ N(0, σ 2 ). Bear in mind that E(Y | X = x) is a deterministic function of x. In other words, the variable x changes with each draw, but after the data have been collected these are no longer random quantities. Thus, for ﬁxed x, y is a random variable generated by . Perhaps we should denote as x to emphasize this, but because is an independent, identically distributed (iid) random variable at each ﬁxed x, this would be excessive. Because of Gaussian additive noise, the distribution of y is completely characterized by its mean and variance.

E(y) = ax + b

V(y) = σ 2

Using the maximum likelihood procedure, we write out the log-likelihood function as

n n 1 L(a, b) = ∑ log N(ax i + b, σ 2 ) ∝ ∑ (y i − ax i − b)2 2σ2 i=1 i=1

Note that we suppressed the terms that are irrelevent to the maximum ﬁnding. Taking the derivative of this with respect to a gives the following equation:

n ∂L(a, b) = 2 x i (b + ax i − y i ) = 0 ∂a ∑ i=1

Likewise, we do the same for the b parameter

n ∂L(a, b) = 2 (b + ax i − y i ) = 0 ∂b ∑ i=1

The following code simulates some data and uses Numpy tools to compute the parameters as shown:

>>> import numpy as np

>>> a = 6;b = 1 # parameters to estimate

>>> x = np.linspace(0,1,100)

>>> y = a\*x + np.random.randn(len(x))+b

>>> p,var\_=np.polyfit(x,y,1,cov=True) # fit data to line

>>> y\_ = np.polyval(p,x) # estimated by linear regression 3.7 Linear Regression

171

Fig. 3.12 The panel on the left shows the data and regression line. The panel on the right shows a histogram of the regression errors

The graph on the left of Fig.3.12 shows the regression line plotted against the data. The estimated parameters are noted in the title. The histogram on the right of Fig.3.12 shows the residual errors in the model. It is always a good idea to inspect the residuals of any regression for normality. These are the differences between the ﬁtted line for each x i value and the corresponding y i value in the data. Note that the x term does not have to be uniformly monotone.

To decouple the deterministic variation from the random variation, we can ﬁx the index and write separate problems of the form

y i = ax i + b + i

where i ∼ N(0, σ 2 ). What could we do with just this one component of the problem? In other words, suppose we had m-samples of this component as in { y i,k } k=1 m . Following the usual procedure, we could obtain estimates of the mean of y i as

m 1 yi ˆ = ∑ yi,k m k=1

However, this tells us nothing about the individual parameters a and b because they are not separable in the terms that are computed, namely, we may have

E(y i ) = ax i + b

but we still only have one equation and the two unknowns, a and b. How about if we consider and ﬁx another component j as in

y j = ax j + b + i

E(y j ) = ax j + b

Then, we have 172

3

Statistics

Fig. 3.13 The ﬁtted and true lines are plotted with the data values. The squares at either end of the solid line show the mean value for each of the data groups shown

so at least now we have two equations and two unknowns and we know how to estimate the left-hand sides of these equations from the data using the estimators yi ˆ and yj ˆ . Let’s see how this works in the code sample below (Fig.3.13):

>>> x0, xn =x[0],x[80]

>>> # generate synthetic data

>>> y\_0 = a\*x0 + np.random.randn(20)+b

>>> y\_1 = a\*xn + np.random.randn(20)+b

>>> # mean along sample dimension

>>> yhat = np.array([y\_0,y\_1]).mean(axis=1)

>>> a\_,b\_=np.linalg.solve(np.array([[x0,1],

...

[xn,1]]),yhat)

Programming Tip

The prior code uses the solve function in the Numpy linalg module, which contains the core linear algebra codes in Numpy that incorporate the battletested LAPACK library.

We can write out the solution for the estimated parameters for this case where x 0 = 0

yi − y0 ˆ aˆ = xi

b ˆ = y0 ˆ 3.7 Linear Regression

173

The expectations and variances of these estimators are the following:

axi E(ˆa) = = a xi

E(b) ˆ = b

2σ2 V(ˆ a) = xi 2

V( b) ˆ = σ 2

Theexpectationsshowthattheestimatorsareunbiased.Theestimator aˆ hasavariance that decreases as larger points x i are selected. That is, it is better to have samples further out along the horizontal axis for ﬁtting the line. This variance quantiﬁes the leverage of those distant points.

Regression From Projection Methods. Let’s see if we can apply our knowledge of projection methods to the general case. In vector notation, we can write the following:

y = ax + b1 +

where 1 is the vector of all ones. Let’s use the inner product notation,

〈 x, y 〉 = E(x T y)

Then, by taking the inner product with some x1

∈

1 ⊥ we obtain,5

〈 y, x 1 〉 = a 〈 x, x 1 〉

Recall that E() = 0. We can ﬁnally solve for a as

y, x1 〈 〉 aˆ = 〈 x, x 1 〉

(3.7.0.1)

That was pretty neat but now we have the mysterious x 1 vector. Where does this come from? If we project x onto the 1 ⊥ , then we get the MMSE approximation to x in the 1 ⊥ space. Thus, we take

x 1 = P 1 ⊥ (x)

Remember that P 1 ⊥ is a projection matrix so the length of x 1 is at most x. This means that the denominator in the aˆ equation above is really just the length of the x vector in the coordinate system of P 1 ⊥ . Because the projection is orthogonal (namely, of minimum length), the Pythagorean theorem gives this length as the following:

〈 x, x 1 〉 2 = 〈 x, x 〉 − 〈 1, x 〉2

5 The space of all vectors, a such that 〈 a, 1 〉 = 0 is denoted 1 ⊥ . 174

3

Statistics

The ﬁrst term on the right is the length of the x vector and last term is the length of x in the coordinate system orthogonal to P 1 ⊥ , namely, that of 1. We can use this geometric interpretationtounderstandwhatisgoingonintypicallinearregressioninmuchmore detail. The fact that the denominator is the orthogonal projection of x tells us that the choice of x 1 has the strongest effect (i.e., largest value) on reducing the variance of a.ˆ That is, the more x is aligned with 1, the worse the variance of a.ˆ This makes intuitive sense because the closer x is to 1, the more constant it is, and we have already seen from our one-dimensional example that distance between the x terms pays off in reduced variance. We already know that aˆ is an unbiased estimator, and, because we chose x 1 deliberately as a projection, we know that it is also of minimum variance. Such estimators are known as minimum-variance unbiased estimators (MVUE).

In the same spirit, let’s examine the numerator of aˆ in Eq.3.7.0.1. We can write x 1 as the following:

x 1 =x− P1 x

where P 1 is projection matrix of x onto the 1 vector. Using this, the numerator of aˆ becomes

〈 y, x 1 〉 = 〈 y, x 〉 − 〈 y, P 1 x 〉

Note that,

1 P 1 = 11T n

so that writing this out explicitly gives

〈 y, P 1 x 〉 = ( y T 1 ) ( 1 T x ) /n = ( ∑ y i ) ( ∑ x i ) /n

and similarly, we have the following for the denominator:

〈 x, P 1 x 〉 = ( x T 1 ) ( 1 T x ) /n = ( ∑ x i ) ( ∑ x i ) /n

So, plugging all of this together gives the following:

xT y − ( ∑ x i )( ∑ y i )/n aˆ = x T x − ( ∑ x i ) 2 /n

with corresponding variance,

∥ x 1 ∥2 V(ˆ a) = σ 2 〈 x, x 1 〉2

σ 2 = ∥ x ∥ 2 − n(x 2 ) 3.7 Linear Regression

175

Using the same approach with

ˆ b gives

⊥ y, x 〈 〉 b ˆ = 〈 1, x ⊥ 〉

(3.7.0.2)

(1) 〉 〈 y, 1 − Px = 〈 1, 1 − P x (1) 〉 xT x( ∑ y i )/n − x T y( ∑ x i )/n = x T x − ( ∑ x i ) 2 /n

(3.7.0.3)

(3.7.0.4)

where

xxT Px = ∥ x ∥2

with variance

2 〈 1 − P x (1), 1 − P x (1) 〉 V( b) ˆ = σ 〈 1, 1 − P x (1) 〉2

σ 2 = 2 n − (nx) 2 ∥ x ∥

Qualifying the Estimates. Our formulas for the variance above include the unknown σ 2 , which we must estimate from the data itself using our plug-in estimates. We can form the residual sum of squares as

RSS b ˆ − yi = ∑ (ˆax i + )2 i

Thus, the estimate of σ 2 can be expressed as

RSS σ ˆ 2 = n−2

wheren isthenumberofsamples.Thisisalsoknownastheresidualmeansquare.The n − 2 represents the degrees of freedom (df). Because we estimated two parameters from the same data we have n − 2 instead of n. Thus, in general, df = n − p, where p is the number of estimated parameters. Under the assumption that the noise is Gaussian, the RSS/σ 2 is chi-squared distributed with n − 2 degrees of freedom. Another important term is the sum of squares about the mean, (a.k.a corrected sum of squares),

SYY = ∑ (y i − ¯y)2

The SYY captures the idea of not using the x i data and just using the mean of the yi data to estimate y. These two terms lead to the R 2 term, 176

3

Statistics

RSS R 2 =1SYY

Note that for perfect regression, R 2 = 1. That is, if the regression gets each y i data point exactly right, then RSS = 0 this term equals one. Thus, this term is used to measure of goodness-of-ﬁt. The stats module in scipy computes many of these terms automatically,

from scipy import stats slope,intercept,r\_value,p\_value,stderr = stats.linregress(x,y)

where the square of the r\_value variable is the R 2 above. The computed p-value is the two-sided hypothesis test with a null hypothesis that the slope of the line is zero. In other words, this tests whether or not the linear regression makes sense for the data for that hypothesis. The Statsmodels module provides a powerful extension to Scipy’s stats module by making it easy to do regression and keeps track of these parameters. Let’s reformulate our problem using the Statsmodels framework by creating a Pandas dataframe for the data,

import statsmodels.formula.api as smf from pandas import DataFrame import numpy as np d = DataFrame({'x':np.linspace(0,1,10)}) # create data d['y'] = a\*d.x+ b + np.random.randn(\*d.x.shape)

Now that we have the input data in the above Pandas dataframe, we can perform the regression as in the following:

results = smf.ols('y ˜x', data=d).fit()

The ∼ symbol is notation for y = ax + b + , where the constant b is implicit in this usage of Statsmodels. The names in the string are taken from the columns in the dataframe. This makes it very easy to build models with complicated interactions between the named columns in the dataframe. We can examine a report of the model ﬁt by looking at the summary,

print (results.summary2()) Results: Ordinary least squares ================================================================= Model: OLS Adj. R-squared: 0.808 Dependent Variable: y AIC: 28.1821 Date: 0000-00-00 00:00 BIC: 00.0000 No. Observations: 10 Log-Likelihood: -12.091 Df Model: 1 F-statistic: 38.86 Df Residuals: 8 Prob (F-statistic): 0.000250 R-squared: 0.829 Scale: 0.82158

------------------------------------------------------------------Coef. Std.Err. t P>|t| [0.025 0.975]

------------------------------------------------------------------Intercept 1.5352 0.5327 2.8817 0.0205 0.3067 2.7637 x 5.5990 0.8981 6.2340 0.0003 3.5279 7.6701 3.7 Linear Regression

177

There is a lot more here than we have discussed so far, but the Statsmodels documentation is the best place to go for complete information about this report. The F-statistic attempts to capture the contrast between including the slope parameter or leaving it off. That is, consider two hypotheses:

H 0 : E(Y | X = x) = b

H 1 : E(Y | X = x) = b + ax

In order to quantify how much better adding the slope term is for the regression, we compute the following:

SYY − RSS F= 2 ˆ σ

The numerator computes the difference in the residual squared errors between including the slope in the regression or just using the mean of the y i values. Once again, if we assume (or can claim asymptotically) that the noise term is Gaussian, ∼ N(0, σ 2 ), then the H 0 hypothesis will follow an F-distribution 6 with degrees of freedom from the numerator and denominator. In this case, F ∼ F(1, n−2). The value of this statistic is reported by Statsmodels above. The corresponding reported probability shows the chance of F exceeding its computed value if H 0 were true. So, the take-home message from all this is that including the slope leads to a much smaller reduction in squared error than could be expected from a favorable draw of n points of this data, under the Gaussian additive noise assumption. This is evidence that including the slope is meaningful for this data.

The Statsmodels report also shows the adjusted R 2 term. This is a correction to the R 2 calculation that accounts for the number of parameters p that the regression is ﬁtting and the sample size n,

RSS/(n − p) Adjusted R 2 = 1 SYY/(n − 1)

This is always lower than R 2 except when p = 1 (i.e., estimating only b). This becomes a better way to compare regressions when one is attempting to ﬁt many parameters with comparatively small n.

Linear Prediction. Using linear regression for prediction introduces some other issues. Recall the following expectation:

ˆ E(Y | X = x) ≈ ˆax + b

where we have determined aˆ and we would certainly compute

ˆ b from the data. Given a new point of interest, xp

ˆ yp ˆ = axp ˆ + b

,

6 The F(m, n) F-distribution has two integer degree-of-freedom parameters, m and n. 178

3

Statistics

as the predicted value for yp ˆ . This is the same as saying that our best prediction for y based on x p is the above conditional expectation. The variance for this is the following:

ˆ V(y p ) = x p 2 V(ˆ a) + V( b) ˆ + 2x p cov(ˆa b)

ˆ Note that we have the covariance above because aˆ and b are derived from the same data. We can work this out below using our previous notation from Eq.3.7.0.1,

x 1 1 T T V { yy T T } x ⊥ x 1 T σ 2 Ix⊥ cov(ˆa b) ˆ = ⊥ = (x x)(1 x ) (x 1 T x)(1 T x ⊥ )

x 1 T x T ⊥ (x − P 1 x) T x ⊥ =σ 2 2 = σ (x 1 T x)(1 x ⊥ ) (x 1 T x)(1 T x ⊥ )

−x T P 1 T T x ⊥ −x T 1 n 11 T x⊥ =σ 2 ⊥ 2 = σ (xT 1 x)(1 x ) (x 1 T x)(1 T x ⊥ )

−x 1 1 −σ 2 x =σ 2 n = (x 1 T T x) ∑ i=1 n (x i 2 − x 2 )

After plugging all this in, we obtain the following:

2 x p 2 − 2x p x + ∥ x ∥ 2 /n V(y p ) = σ ∥ x ∥ 2 − nx2

where, in practice, we use the plug-in estimate for the σ 2 .

There is an important consequence for the conﬁdence interval for y p . We cannot simply use the square root of V(y p ) to form the conﬁdence interval because the model includes the extra noise term. In particular, the parameters were computed using a set of statistics from the data, but now must include different realizations for the noise term for the prediction part. This means we have to compute

2 η = V(y p ) + σ2

Then, the 95% conﬁdence interval y p ∈ (y p − 2 η ˆ , y p + 2 η ˆ ) is the following:

P(y p − 2 η ˆ < y p < y p + 2 η ˆ ) ≈ P(−2 < N(0, 1) < 2) ≈ 0.95

where η ˆ comes from substituting the plug-in estimate for σ.

3.7.1 Extensions to Multiple Covariates

With all the machinery we have, it is a short notational hop to consider multiple regressors as in the following: 3.7 Linear Regression

179

Y = X β +

with the usual E() = 0 and V() = σ 2 I. Thus, X is a n × p full rank matrix of regressors and Y is the n-vector of observations. Note that the constant term has been incorporated into X as a column of ones. The corresponding estimated solution for β is the following:

β = (X T X) −1 X T Y

with corresponding variance,

V( β ) = σ 2 (X T X)−1

and with the assumption of Gaussian errors, we have

β ∼ N( β , σ 2 (X T X) −1 )

The unbiased estimate of σ 2 is the following:

1 2 σ ˆ 2 = ∑ i n−p

where ˆ = X β − Y is the vector of residuals. Tukey christened the following matrix as the hat matrix (a.k.a. inﬂuence matrix):

V = X(X T X) −1 XT

because it maps Y into

ˆ Y,

Y ˆ = VY

As an exercise you can check that V is a projection matrix. Note that that matrix is solely a function of X. The diagonal elements of V are called the leverage values and are contained in the closed interval [ 1/n, 1 ] . These terms measure distance between the values of x i and the mean values over the n observations. Thus, the leverage terms dependonlyonX.Thisisthegeneralizationofourinitialdiscussionofleveragewhere we had multiple samples at only two x i points. Using the hat matrix, we can compute the variance of each residual, e i = yˆ − y i as

V(e i ) = σ 2 (1 − v i )

where v i = V i,i . Given the abovementioned bounds on v i , these are always less than σ 2 .

Degeneracy in the columns of X can become a problem. This is when two or more of the columns become co-linear. We have already seen this with our single regressor example wherein x close to 1 was bad news. To compensate for this effect 180

3

Statistics

we can load the diagonal elements and solve for the unknown parameters as in the following:

β = (X T X + αI) −1 X T Y

where α > 0 is a tunable hyper-parameter. This method is known as ridge regression and was proposed in 1970 by Hoerl and Kenndard. It can be shown that this is the equivalent to minimizing the following objective:

∥ Y − X β∥ 2 + α ∥β∥2

In other words, the length of the estimated β is penalized with larger α. This has the effect of stabilizing the subsequent inverse calculation and also providing a means to trade bias and variance, which we will discuss at length in Sect.4.8.

Interpreting Residuals. Our model assumes an additive Gaussian noise term. We can check the voracity of this assumption by examining the residuals after ﬁtting. The residuals are the difference between the ﬁtted values and the original data

i ˆ = axi ˆ + b ˆ − yi

While the p-value and the F-ratio provide some indication of whether or not computing the slope of the regression makes sense, we can get directly at the key assumption of additive Gaussian noise.

For sufﬁciently small dimensions, the scipy.stats.probplot we discussed in the last chapter provides quick visual evidence one way or another by plotting the standardized residuals,

ei r i = σ ˆ √ 1 − vi

The other part of the iid assumption implies homoscedasticity (all r i have equal variances). Under the additive Gaussian noise assumption, the e i should also be distributed according to N(0, σ 2 (1 − v i )). The normalized residuals r i should then be distributed according to N(0, 1). Thus, the presence of any r i ∈ / [ −1.96, 1.96 ] should not be common at the 5% signiﬁcance level and is thereby breeds suspicion regarding the homoscedasticity assumption.

The Levene test in scipy.stats.leven tests the null hypothesis that all the variances are equal. This basically checks whether or not the standardized residuals vary across x i more than expected. Under the homoscedasticity assumption, the variance should be independent of x i . If not, then this is a clue that there is a missing variable in the analysis or that the variables themselves should be transformed (e.g., using the log function) into another format that can reduce this effect. Also, we can use weighted least-squares instead of ordinary least-squares.

Variable Scaling. It is tempting to conclude in a multiple regression that small coefﬁcients in any of the β terms implies that those terms are not important. However, simple unit conversions can cause this effect. For example, if one of the regressors 3.7 Linear Regression

181

Fig. 3.14 The point on the right has outsized inﬂuence in this data because it is the only one used to determine the slope of the ﬁtted line

is in units of kilometers and the others are in meters, then just the scale factor can give the impression of outsized or under-sized effects. The common way to account for this is to scale the regressors so that

x ¯x x ′ =

σx

This has the side effect of converting the slope parameters into correlation coefﬁcients, which is bounded by ±1.

Inﬂuential Data. We have already discussed the idea of leverage. The concept of inﬂuence combines leverage with outliers. To understand inﬂuence, consider Fig.3.14.

The point on the right in Fig.3.14 is the only one that contributes to the calculation of the slope for the ﬁtted line. Thus, it is very inﬂuential in this sense. Cook’s distance is a good way to get at this concept numerically. To compute this, we have to compute the j th component of the estimated target variable with the i th point deleted. We call this yj(i) ˆ . Then, we compute the following:

( yj ˆ − yj(i) ˆ )2 ∑j D i = p/n ∑ j ( yj ˆ − y j )2

where, as before, p is the number of estimated terms (e.g., p = 2 in the bivariate case). This calculation emphasizes the effect of the outlier by predicting the target variable with and without each point. In the case of Fig.3.14, losing any of the points on the left cannot change the estimated target variable much, but losing the single point on the right surely does. The point on the right does not seem to be an outlier (it is on the ﬁtted line), but this is because it is inﬂuential enough to rotate the line to align with it. Cook’s distance helps capture this effect by leaving each sample out and re-ﬁtting the remainder as shown in the last equation. Figure3.15 shows the 182

3

Statistics

Fig. 3.15 The calculated Cook’s distance for the data in Fig.3.14

calculated Cook’s distance for the data in Fig.3.14, showing that the data point on the right (sample index 5) has outsized inﬂuence on the ﬁtted line. As a rule of thumb, Cook’s distance values that are larger than one are suspect.

As another illustration of inﬂuence, consider Fig.3.16 which shows some data that nicely line up, but with one outlier (ﬁlled black circle) in the upper panel. The lower panel shows so-computed Cook’s distance for this data and emphasizes the presence of the outlier. Because the calculation involves leaving a single sample out and re-calculating the rest, it can be a time-consuming operation suitable to relatively small datasets. There is always the temptation to downplay the importance of outliers because they conﬂict with a favored model, but outliers must be carefully examined to understand why the model is unable to capture them. It could be something as simple as faulty data collection, or it could be an indication of deeper issues that have been overlooked. The following code shows how Cook’s distance was compute for Figs.3.15 and 3.16.

>>> fit = lambda i,x,y: np.polyval(np.polyfit(x,y,1),i)

>>> omit = lambda i,x: ([k for j,k in enumerate(x) if j !=i])

>>> def cook\_d(k):

... num = sum((fit(j,omit(k,x),omit(k,y))-fit(j,x,y))\*\*2 for j in x) ...

den

=

sum((y-np.polyval(np.polyfit(x,y,1),x))\*\*2/len(x)\*2)

... return num/den ...

Programming Tip

The function omit sweeps through the data and excludes the i th data element. The embedded enumerate function associates every element in the iterable with its corresponding index. 3.8 Maximum A-Posteriori

183

Fig. 3.16 The upper panel shows data that ﬁt on a line and an outlier point (ﬁlled black circle). The lower panel shows the calculated Cook’s distance for the data in upper panel and shows that the tenth point (i.e., the outlier) has disproportionate inﬂuence

3.8 Maximum A-Posteriori

We saw with maximum likelihood estimation how we could use the principle of maximum likelihood to derive a formula of the data that would estimate the underlying parameters (say, θ). Under that method, the parameter was ﬁxed, but unknown. If we change our perspective slightly and consider the underlying parameter as a random variable in its own right, this leads to additional ﬂexibility in estimation. This method is the simplest of the family of Bayesian statistical methods and is most closely related to maximum likelihood estimation. It is very popular in communications and signal processing and is the backbone of many important algorithms in those areas.

Given that the parameter θ is also a random variable, it has a joint distribution with the other random variables, say, f (x, θ). Bayes’ theorem gives the following:

P(x | θ)P(θ) P(θ | x) = P(x)

The P(x | θ) term is the usual likelihood term we have seen before. The term in the denominator is prior probability of the data x and it explicitly makes a very powerful claim: even before collecting or processing any data, we know what the probability of that data is. The P(θ) is the prior probability of the parameter. In other words, regardless of the data that is collected, this is the probability of the parameter itself.

In a particular application, whether or not you feel justiﬁed making these claims is something that you have to reconcile for yourself and the problem at hand. There are many persuasive philosophical arguments one way or the other, but the main thing to keep in mind when applying any method is whether or not the assumptions are reasonable for the problem at hand.

However, for now, let’s just assume that we somehow have P(θ) and the next step is the maximizing of this expression over the θ. Whatever results from that maximiza- 184

3

Statistics

tion is the maximum a-posteriori (MAP) estimator for θ. Because the maximization takes place with respect to θ and not x, we can ignore the P(x) part. To make things concrete, let us return to our original coin-ﬂipping problem. From our earlier analysis, we know that the likelihood function for this problem is the following:

(θ) := θ k (1 − θ)(n−k)

where the probability of the coin coming up heads is θ. The next step is the prior probability, P(θ). For this example, we will choose the β (6, 6) distribution (shown in the top left panel of Fig.3.17). The β family of distributions is a gold mine because it allows for a wide variety of distributions using few input parameters. Now that we have all the ingredients, we turn to maximizing the posterior function, P(θ | x). Because the logarithm is convex, we can use it to make the maximization process easier by converting the product to a sum without changing the extrema that we are looking for. Thus, we prefer to work with the logarithm of P(θ | x) as in the following:

L := log P(θ | x) = log (θ) + log P(θ) − log P(x)

This is tedious to do by hand and therefore an excellent job for Sympy.

>>> import sympy

>>> from sympy import stats as st

>>> from sympy.abc import p,k,n

# setup objective function using sympy.log

>>> obj=sympy.expand\_log(sympy.log(p\*\*k\*(1-p)\*\*(n-k)\* st.density(st.Beta('p',6,6))(p))) # use calculus to maximize objective

>>> sol=sympy.solve(sympy.simplify(sympy.diff(obj,p)),p)[0]

>>> sol

(k + 5)/(n + 10)

which means that our MAP estimator of θ is the following:

k 5 + θ M AP ˆ = n + 10

where k is the number of heads in the sample. This is obviously a biased estimator of θ,

5 nθ + E( θ M AP ˆ ) = = θ 10 + n

But is this bias bad? Why would anyone want a biased estimator? Remember that we constructed this entire estimator using the idea of the prior probability of P(θ) which favors (biases!) the estimate according to the prior. For example, if θ = 1/2, the MAP estimator evaluates to θ M AP ˆ = 1/2. No bias there! This is because the peak of the prior probability is at θ = 1/2. 3.8 Maximum A-Posteriori

185

To compute the corresponding variance for this estimator, we need this intermediate result,

25 10nθ nθ((n 1)p 1) + + + E( θ M AP 2 ˆ ) = (10 + n)2

which gives the following variance:

n(1 θ)θ V( θ M AP ˆ ) = (n + 10)2

Let’s pause and compare this to our previous maximum likelihood (ML) estimator shown below:

n 1 k θ ML ˆ = ∑ X i = n n i=1

As we discussed before, the ML-estimator is unbiased with the following variance:

θ(1 θ) V( θ ML ˆ ) = n

How does this variance compare to that of the MAP? The ratio of the two is the following:

ˆ V( θ M AP ) n2 = θ ML ˆ ) (n + 10)2 V(

This ratio shows that the variance for the MAP estimator is smaller than that of the ML-estimator. This is payoff for having a biased MAP estimator—it requires fewer samples to estimate if the underlying parameter is consistent with the prior probability. If not, then it will take more samples to pull the estimator away from the bias. In the limit as n → ∞, the ratio goes to one. This means that the beneﬁt of the reduced variance vanishes with enough samples.

The above discussion admits a level of arbitrariness via the prior distribution. We don’t have to choose just one prior, however. The following shows how we can use the previous posterior distribution as the prior for the next posterior distribution:

P(xk + 1 | θ)P(θ | x k ) P(θ | x k + 1 ) = P(x k + 1 )

This is a very different strategy because we are using every data sample x k as a parameter for the posterior distribution instead of lumping all the samples together in a summation (this is where we got the k term in the prior case). This case is much harder to analyze because now every incremental posterior distribution is itself a random function because of the injection of the x random variable. On the other hand, this is more in line with more general Bayesian methods because it is clear that the output of this estimation process is a posterior distribution function, not just a single parameter estimate. 186

3

Statistics

Fig. 3.17 The prior probability is the β (6, 6) distribution shown in the top left panel. The dots near the peaks of each of the subgraphs indicate the MAP estimate at that frame

Figure3.17 illustrates this method. The graph in the top row, far left shows the prior probability ( β (6, 6)) and the dot on the top shows the most recent MAP estimate for θ. Thus, before we obtain any data, the peak of the prior probability is the estimate. The next graph to right shows the effect of x 0 = 0 on the incremental prior probability. Note that the estimate has barely moved to the left. This is because the inﬂuence of the data has not caused the prior probability to drift away from the original β (6, 6)distribution. The ﬁrst two rows of the ﬁgure all have x k = 0 just to illustrate how far left the original prior probability can be moved by those data. The dots on the tops of the subgraphs show how the MAP estimate changes frame-by-frame as more data is incorporated. The remaining graphs, proceeding top-down and left-to-right, show the incremental change in the prior probability for x k = 1. Again, this shows how far to the right the estimate can be pulled from where it started. For this example, there are an equal number of x k = 0 and x k = 1 data, which correspond to θ = 1/2. 3.8 Maximum A-Posteriori

187

Programming Tip

The following is a quick paraphrase of how Fig.3.17 was constructed. The ﬁrst step is to recursively create the posteriors from the data. Note the example data is sorted to make the progression easy to see as a sequence.

from sympy.abc import p,x from scipy.stats import density, Beta, Bernoulli prior = density(Beta('p',6,6))(p) likelihood=density(Bernoulli('x',p))(x) data = (0,0,0,0,0,0,0,1,1,1,1,1,1,1,1) posteriors = [prior] for i in data:

posteriors.append(posteriors[-1]\*likelihood.subs(x,i))

With the posteriors in hand, the next step is to compute the peak values at each frame using the fminbound function from Scipy’s optimize module.

pvals = linspace(0,1,100) mxvals = [] for i,j in zip(ax.flat,posteriors):

i.plot(pvals,sympy.lambdify(p,j)(pvals),color='k')

mxval = fminbound(sympy.lambdify(p,-j),0,1)

mxvals.append(mxval)

h = i.axis()[-1]

i.axis(ymax=h\*1.3) i.plot(mxvals[-1],h\*1.2,'ok') i.plot(mxvals[:-1],[h\*1.2]\*len(mxvals[:-1]),'o')

Figure3.18 is the same as Fig.3.17 except that the initial prior probability is the β (1.3, 1.3)-distribution, which has a wider lobe that the β (6, 6)-distribution. As shown in the ﬁgure, this prior has the ability to be swayed more violently one way or the other based on the x k data that is incorporated. This means that it can more quickly adapt to data that is not so consistent with the initial prior and thus does not require a large amount of data in order to unlearn the prior probability. Depending on the application, the ability to unlearn the prior probability or stick with it is a design problem for the analyst. In this example, because the data are representative of a θ = 1/2 parameter, both priors eventually settle on an estimated posterior that is about the same. However, if this had not been the case (θ = ̸ 1/2), then the second prior would have produced a better estimate for the same amount of data.

Because we have the entire posterior density available, we can compute something that is closely related to the conﬁdence interval we discussed earlier, except in this situation, given the Bayesian interpretation, it is called a credible interval or credible set. The idea is that we want to ﬁnd a symmetric interval around the peak that accounts 188

3

Statistics

Fig. 3.18 For this example, the prior probability is the β (1.3, 1.3) distribution, which has a wider main lobe than the β (6, 6) distribution. The dots near the peaks of each of the subgraphs indicate the MAP estimate at that frame

for 95% (say) of the posterior density. This means that we can then say the probability that the estimated parameter is within the credible interval is 95%. The computation requires signiﬁcant numerical processing because even though we have the posterior density in hand, it is hard to integrate analytically and requires numerical quadrature (see Scipy’s integrate module). Figure3.19 shows extent of the interval and the shaded region under the posterior density that accounts for 95%.

3.9 Robust Statistics

We considered maximum likelihood estimation (MLE) and maximum a-posteriori (MAP) estimation and in each case we started out with a probability density function 3.9 Robust Statistics

189

Fig. 3.19 The credible interval in Bayesian maximum a-posteriori is the interval corresponding to the shaded region in the posterior density

of some kind and we further assumed that the samples were identically distributed and independent (iid). The idea behind robust statistics [3] is to construct estimators that can survive the weakening of either or both of these assumptions. More concretely, suppose you have a model that works great except for a few outliers. The temptation is to just ignore the outliers and proceed. Robust estimation methods provide a disciplined way to handle outliers without cherry-picking data that works for your favored model.

The Notion of Location. The ﬁrst notion we need is location, which is a generalization of the idea of central value. Typically, we just use an estimate of the mean for this, but we will see later why this could be a bad idea. The general idea of location satisﬁes the following requirements. Let X be a random variable with distribution F, and let θ(X) be some descriptive measure of F. Then θ(X) is said to be a measure of location if for any constants a and b, we have the following:

θ(X + b) = θ(X) + b

θ(−X) = −θ(X)

X ≥ 0 ⇒ θ(X) ≥ 0

θ(aX) = aθ(X)

(3.9.0.1) (3.9.0.2) (3.9.0.3) (3.9.0.4)

The ﬁrst condition is called location equivariance (or shift-invariance in signal processing lingo). The fourth condition is called scale equivariance, which means that the units that X is measured in should not effect the value of the location estimator. These requirements capture the intuition of centrality of a distribution, or where most of the probability mass is located.

1 For example, the sample mean estimator is μ ˆ = n ∑ X i . The ﬁrst requirement is 1 1 obviously satisﬁed as μ ˆ = n ∑ (X i + b) = b + n ∑ X i = b + μ ˆ . Let us consider the 1 second requirement: μ ˆ = n ∑ −X i = − μ ˆ . Finally, the last requirement is satisﬁed 1 with μ ˆ = n ∑ aX i = a μ ˆ . 190

3

Statistics

Robust Estimation and Contamination. Now that we have the generalized location of centrality embodied in the location parameter, what can we do with it? Previously, we assumed that our samples were all identically distributed. The key idea is that the samples might be actually coming from a single distribution that is contaminated by another nearby distribution, as in the following:

F(X) = G(X) + (1 − )H(X)

where randomly toggles between zero and one. This means that our data samples { X i } actually derived from two separate distributions, G(X) and H(X). We just don’t know how they are mixed together. What we really want is an estimator that captures the location of G(X) in the face of random intermittent contamination by H(X). For example, it may be that this contamination is responsible for the outliers in a model that otherwise works well with the dominant F distribution. It can get even worse than that because we don’t know that there is only one contaminating H(X) distribution out there. There may be a whole family of distributions that are contaminating G(X). This means that whatever estimators we construct have to be derived from a more generalized family of distributions instead of from a single distribution, as the maximum likelihood method assumes. This is what makes robust estimation so difﬁcult—it has to deal with spaces of function distributions instead of parameters from a particular probability distribution.

Generalized Maximum Likelihood Estimators. M-estimators are generalized maximum likelihood estimators. Recall that for maximum likelihood, we want to maximize the likelihood function as in the following:

L μ (x i ) = ∏ f 0 (x i − μ )

and then to ﬁnd the estimator μ ˆ so that

μ ˆ = arg max L μ (x i ) μ

So far, everything is the same as our usual maximum likelihood derivation except for the fact that we don’t assume a speciﬁc f 0 as the distribution of the { X i } . Making the deﬁnition of

ρ = − log f0

we obtain the more convenient form of the likelihood product and the optimal μ ˆ as

μ ˆ = arg min ∑ ρ (x i − μ ) μ

If ρ is differentiable, then differentiating this with respect to μ gives

∑ ψ (x i − μ ˆ ) = 0

(3.9.0.5) 3.9 Robust Statistics

191

with ψ = ρ ′ , the ﬁrst derivative of ρ , and for technical reasons we will assume that ψ is increasing. So far, it looks like we just pushed some deﬁnitions around, but the key idea is we want to consider general ρ functions that may not be maximum likelihood estimators for any distribution. Thus, our focus is now on uncovering the nature of μ ˆ .

Distribution of M-Estimates. For a given distribution F, we deﬁne μ 0 = μ (F) as the solution to the following:

E F ( ψ (x − μ 0 )) = 0

v It is technical to show, but it turns out that μ ˆ ∼ N( μ 0 , n ) with

EF ( ψ (x − μ 0 ) 2 ) v= (E F ( ψ ′ (x − μ 0 )))2

Thus, we can say that μ ˆ is asymptotically normal with asymptotic value μ 0 and asymptotic variance v. This leads to the efﬁciency ratio which is deﬁned as the following:

v0 Eff( μ ˆ ) = v

where v 0 is the asymptotic variance of the MLE and measures how near μ ˆ is to the optimum. In other words, this provides a sense of how much outlier contamination costs in terms of samples. For example, if for two estimates with asymptotic variances v 1 and v 2 , we have v 1 = 3v 2 , then ﬁrst estimate requires three times as many observations to obtain the same variance as the second. Furthermore, for the sample 1 mean (i.e., μ ˆ = n ∑ X i ) with F = N, we have ρ = x 2 /2 and ψ = x and also ψ ′ = 1. Thus, we have v = V(x). Alternatively, using the sample median as the estimator for the location, we have v = 1/(4 f ( μ 0 ) 2 ). Thus, if we have F = N(0, 1), for the sample median, we obtain v = 2π/4 ≈ 1.571. This means that the sample median takes approximately 1.6 times as many samples to obtain the same variance for the location as the sample mean. The sample median is far more immune to the effects of outliers than the sample mean, so this gives a sense of how much this robustness costs in samples.

M-EstimatesasWeightedMeans.OnewaytothinkaboutM-estimatesisaweighted means. Operationally, this means that we want weight functions that can circumscribe the inﬂuence of the individual data points, but, when taken as a whole, still provide good estimated parameters. Most of the time, we have ψ (0) = 0 and ψ ′ (0) exists so that ψ is approximately linear at the origin. Using the following deﬁnition:

(x)/x if x = ̸ 0 W(x) = ψ ′ { ψ (x) if x = 0 192

3

Statistics

We can write our Eq.3.9.0.5 as follows:

∑ W(x i − μ ˆ )(x i − μ ˆ ) = 0

(3.9.0.6)

Solving this for μ ˆ yields the following,

xi ∑ wi μ ˆ = wi ∑

where w i = W(x i − μ ˆ ). This is not practically useful because the w i contains μ ˆ , which is what we are trying to solve for. The question that remains is how to pick the ψ functions. This is still an open question, but the Huber functions are a wellstudied choice.

Huber Functions. The family of Huber functions is deﬁned by the following:

x 2 if | x | ≤ k (x) = ρk { 2k | x | − k 2 if | x | > k

with corresponding derivatives 2 ψ k (x) with

x if | x | ≤ k (x) = ψk { sgn(x)k if | x | > k

where the limiting cases k → ∞ and k → 0 correspond to the mean and median, respectively. To see this, take ψ ∞ = x and therefore W(x) = 1 and thus the deﬁning Eq.3.9.0.6 results in

n ∑ (x i − μ ˆ ) = 0 i=1

1 and then solving this leads to μ ˆ = n ∑ x i . Note that choosing k = 0 leads to the sample median, but that is not so straightforward to solve for. Nonetheless, Huber functions provide a way to move between two extremes of estimators for location (namely, the mean vs. the median) with a tunable parameter k. The W function corresponding to Huber’s ψ is the following:

k W k (x) = min 1, { | x | }

Figure3.20 shows the Huber weight function for k = 2 with some sample points. The idea is that the computed location, μ ˆ is computed from Eq.3.9.0.6 to lie somewhere in the middle of the weight function so that those terms (i.e., insiders) have their values fully reﬂected in the location estimate. The black circles are the outliers that 3.9 Robust Statistics

193

Fig. 3.20 This shows the Huber weight function, W 2 (x) and some cartoon data points that are insiders or outsiders as far as the robust location estimate is concerned

have their values attenuated by the weight function so that only a fraction of their presence is represented in the location estimate.

Breakdown Point. So far, our discussion of robustness has been very abstract. A more concrete concept of robustness comes from the breakdown point. In the simplest terms, the breakdown point describes what happens when a single data point in an estimator is changed in the most damaging way possible. For example, suppose we have the sample mean, μ ˆ = ∑ x i /n, and we take one of the x i points to be inﬁnite. What happens to this estimator? It also goes inﬁnite. This means that the breakdown point of the estimator is 0%. On the other hand, the median has a breakdown point of 50%, meaning that half of the data for computing the median could go inﬁnite without affecting the median value. The median is a rank statistic that cares more about the relative ranking of the data than the values of the data, which explains its robustness.

The simplest but still formal way to express the breakdown point is to take n data points, D = { (x i , y i ) } . Suppose T is a regression estimator that yields a vector of regression coefﬁcients, θ,

T (D) = θ

Likewise, consider all possible corrupted samples of the data D ′ . The maximum bias caused by this contamination is the following:

bias m = sup ∥ T (D ′ ) − T (D) ∥ D′

where the sup sweeps over all possible sets of m-contaminated samples. Using this, the breakdown point is deﬁned as the following:

m biasm m min = : → ∞ { n } 194

3

Statistics

For example, in our least-squares regression, even one point at inﬁnity causes an inﬁnite T . Thus, for least-squares regression, m = 1/n. In the limit n → ∞, we have m → 0.

Estimating Scale. In robust statistics, the concept of scale refers to a measure of the dispersion of the data. Usually, we use the estimated standard deviation for this, but this has a terrible breakdown point. Even more troubling, in order to get a good estimate of location, we have to either somehow know the scale ahead of time, or jointly estimate it. None of these methods have easy-to-compute closed-form solutions and must be computed numerically.

The most popular method for estimating scale is the median absolute deviation

MAD = Med( | x − Med(x) | )

In words, take the median of the data x and then subtract that median from the data itself, and then take the median of the absolute value of the result. Another good dispersion estimate is the interquartile range,

IQR = x (n−m + 1) − x(n)

where m = [ n/4 ] . The x (n) notation means the n th data element after the data have been sorted. Thus, in this notation, max(x) = x (n) . In the case where x ∼ N( μ , σ 2 ), then MAD and IQR are constant multiples of σ such that the normalized MAD is the following:

MAD MADN(x) = 0.675

The number comes from the inverse CDF of the normal distribution corresponding to the 0.75 level. Given the complexity of the calculations, jointly estimating both location and scale is a purely numerical matter. Fortunately, the Statsmodels module has many of these ready to use. Let’s create some contaminated data in the following code:

import statsmodels.api as sm from scipy import stats data=np.hstack([stats.norm(10,1).rvs(10), stats.norm(0,1).rvs(100)])

These data correspond to our model of contamination that we started this section with. As shown in the histogram in Fig.3.21, there are two normal distributions, one centered neatly at zero, representing the majority of the samples, and another coming less regularly from the normal distribution on the right. Notice that the group of infrequent samples on the right separates the mean and median estimates (vertical dotted and dashed lines). In the absence of the contaminating distribution on the right, the standard deviation for this data should be close to one. However, the usual non-robust estimate for standard deviation (np.std) comes out to approximately three. Using 3.9 Robust Statistics

195

Fig. 3.21 Histogram of sample data. Notice that the group of infrequent samples on the right separates the mean and median estimates indicated by the vertical lines

theMADNestimator(sm.robust.scale.mad(data))weobtainapproximately1.25. Thus, the robust estimate of dispersion is less moved by the presence of the contaminating distribution.

The generalized maximum likelihood M-estimation extends to joint scale and location estimation using Huber functions. For example,

huber = sm.robust.scale.Huber() loc,scl=huber(data)

which implements Huber’s proposal two method of joint estimation of location and scale.Thiskindofestimationisthekeyingredienttorobustregressionmethods,many ofwhichareimplementedinStatsmodelsin statsmodels.formula.api.rlm.The corresponding documentation has more information.

3.10 Bootstrapping

As we have seen, it can be very difﬁcult or impossible to determine the probability density distribution of the estimator of some quantity. The idea behind the bootstrap is that we can use computation to approximate these functions which would otherwise be impossible to solve for analytically.

Let’s start with a simple example. Suppose we have the following set of random variables, { X 1 , X 2 , . . . , X n } where each X k ∼ F. In other words the samples are all drawn from the same unknown distribution F. Having run the experiment, we thereby obtain the following sample set:

{ x 1 , x 2 , . . . , x n } 196

3

Statistics

The sample mean is computed from this set as

n 1 ¯x = ∑ xi n i=1

The next question is how close is the sample mean to the true mean, θ = E F (X). Note that the second central moment of X is as follows:

μ 2 (F) := E F (X 2 ) − (E F (X))2

The standard deviation of the sample mean, ¯x, given n samples from an underlying distribution F, is the following:

σ(F) = ( μ 2 (F)/n)1/2

Unfortunately, because we have only the set of samples { x 1 , x 2 , . . . , x n } and not F itself, we cannot compute this and instead must use the estimated standard error,

¯σ ¯μ2 = ( /n)1/2

¯μ2 where = ∑ (x i − ¯x) 2 /(n−1), which is the unbiased estimate of μ 2 (F). However, this is not the only way to proceed. Instead, we could replace F by some estimate, F ˆ obtained as a piece-wise function of { x 1 , x 2 , . . . , x n } by placing probability mass 1/n on each x i . With that in place, we can compute the estimated standard error as the following:

ˆ σ B ˆ = ( μ 2 ( F)/n)1/2

which is called the bootstrap estimate of the standard error. Unfortunately, the story effectively ends here. In even a slightly more general setting, there is no clean formula ˆ σ(F) within which F can be swapped for F. This is where the computer saves the day. We actually do not need to know the formula σ(F) because we can compute it using a resampling method. The key idea is to sample with replacement from { x 1 , x 2 , . . . , x n } . The new set of n independent draws (with replacement) from this set is the bootstrap sample,

y ∗ = { x 1 ∗ , x 2 ∗ , . . . , x n ∗ }

The Monte Carlo algorithm proceeds by ﬁrst by selecting a large number of bootstrap samples, { y k ∗ } , then computing the statistic on each of these samples, and then computing the sample standard deviation of the results in the usual way. Thus, the bootstrap estimate of the statistic θ is the following:

1 ˆ θ ˆ ∗ = ∑ θ ∗ (k) B B k 3.10 Bootstrapping

197

Fig. 3.22 The β (3, 2) distribution and the histogram that approximates it

with the corresponding square of the sample standard deviation as

1 ˆ σ ˆ 2 = ∑ ( θ ∗ ˆ (k) − θ B ∗ )2 B B−1 k

The process is much simpler than the notation implies. Let’s explore this with a simple example using Python. The next block of code sets up some samples from a β (3, 2) distribution,

>>> import numpy as np

>>> from scipy import stats

>>> rv = stats.beta(3,2)

>>> xsamples = rv.rvs(50)

Because this is simulation data, we already know that the mean is μ 1 = 3/5 and the ¯σ standard deviation of the sample mean for n = 50 is = √ 2/50, which we will verify later.

Figure3.22 shows the β (3, 2) distribution and the corresponding histogram of the ˆ samples. The histogram represents F and is the distribution we sample from to obtain ˆ the bootstrap samples. As shown, the F is a pretty crude estimate for the F density (smooth solid line), but that’s not a serious problem insofar as the following bootstrap ˆ estimates are concerned. In fact, the approximation F has a natural tendency to pull toward the bulk of probability mass. This is a feature, not a bug; and is the underlying mechanism that explains bootstrapping, but the formal proofs that exploit this basic idea are far out of our scope here. The next block generates the bootstrap samples

>>> yboot = np.random.choice(xsamples,(100,50))

>>> yboot\_mn = yboot.mean() 198

3

Statistics

Fig. 3.23 For each bootstrap draw, we compute the sample mean. This is the histogram of those sample means that will be used to compute the bootstrap estimate of the standard deviation

and the bootstrap estimate is therefore,

>>> np.std(yboot.mean(axis=1)) # approx sqrt(1/1250) 0.025598763883825818

Figure3.23 shows the distribution of computed sample means from the bootstrap samples. As promised, the next block shows how to use sympy.stats to compute the β (3, 2) parameters we quoted earlier.

>>> import sympy as S

>>> import sympy.stats

>>> for i in range(50): # 50 samples

... # load sympy.stats Beta random variables ... # into global namespace using exec ... execstring = "x%d = S.stats.Beta('x'+str(%d),3,2)"%(i,i) ... exec(execstring) ...

>>> # populate xlist with the sympy.stats random variables

>>> # from above

>>> xlist = [eval('x%d'%(i)) for i in range(50) ]

>>> # compute sample mean

>>> sample\_mean = sum(xlist)/len(xlist)

>>> # compute expectation of sample mean

>>> sample\_mean\_1 = S.stats.E(sample\_mean)

>>> # compute 2nd moment of sample mean

>>> sample\_mean\_2 = S.stats.E(S.expand(sample\_mean\*\*2))

>>> # standard deviation of sample mean

>>> # use sympy sqrt function

>>> sigma\_smn = S.sqrt(sample\_mean\_2-sample\_mean\_1\*\*2) # sqrt(2)/50

>>> print(sigma\_smn)

sqrt(-9\*hyper((4,), (6,), 0)\*\*2/25 + hyper((5,), (7,), 0)/125 + 49/(20000\*beta (3, 2)\*\*2)) 3.10 Bootstrapping

199

Programming Tip

Using the exec function enables the creation of a sequence of Sympy random variables. Sympy has the var function which can automatically create a sequence of Sympy symbols, but there is no corresponding function in the statistics module to do this for random variables.

Example. Recall the delta method from Sect.3.4.2. Suppose we have a set of Bernoulli coin-ﬂips (X i ) with probability of head p. Our maximum likelihood estimator of p is pˆ = ∑ X i /n for n ﬂips. We know this estimator is unbiased with E( p)ˆ = p and V( p)ˆ = p(1 − p)/n. Suppose we want to use the data to estimate the variance of the Bernoulli trials (V(X) = p(1− p)). By the notation the delta method, g(x) = x(1−x). By the plug-in principle, our maximum likelihood estimator of this variance is then ˆp(1 − ˆp). We want the variance of this quantity. Using the results of the delta method, we have

V(g( ˆp)) = (1 − 2 p)2 ˆ V( p)ˆ

ˆp(1 − p)ˆ V(g( ˆp)) = (1 − 2 p)2 ˆ n

Let’s see how useful this is with a short simulation.

>>> from scipy import stats

>>> import numpy as np

>>> p= 0.25 # true head-up probability

>>> x = stats.bernoulli(p).rvs(10)

>>> print(x)

[0 0 0 0 0 0 1 0 0 0]

The maximum likelihood estimator of p is pˆ = ∑ X i /n,

>>> phat = x.mean()

>>> print(phat)

0.1

Then, plugging this into the delta method approximant above,

>>> print((1-2\*phat)\*\*2\*(phat)\*\*2/10) 0.0006400000000000003

Now, let’s try this using the bootstrap estimate of the variance

>>> phat\_b=np.random.choice(x,(50,10)).mean(1)

>>> print(np.var(phat\_b\*(1-phat\_b)))

0.0050490000000000005 200

3

Statistics

This shows that the delta method’s estimated variance is different from the bootstrap method, but which one is better? For this situation we can solve for this directly using Sympy

>>> import sympy as S

>>> from sympy.stats import E, Bernoulli

>>> xdata =[Bernoulli(i,p) for i in S.symbols('x:10')]

>>> ph = sum(xdata)/float(len(xdata))

>>> g = ph\*(1-ph)

Programming Tip

The argument in the S.symbols(’x:10’) function returns a sequence of Sympy symbols named x1,x2 and so on. This is shorthand for creating and naming each symbol sequentially.

Note that g is the g( p)ˆ = ˆp(1 − p)ˆ whose variance we are trying to estimate. Then, we can plug in for the estimated pˆ and get the correct value for the variance,

>>> print(E(g\*\*2) - E(g)\*\*2) 0.00442968750000000

This case is generally representative—the delta method tends to underestimate the variance and the bootstrap estimate is better here.

3.10.1 Parametric Bootstrap

In the previous example, we used the { x 1 , x 2 , . . . , x n } samples themselves as the ˆ basis for F by weighting each with 1/n. An alternative is to assume that the samples come from a particular distribution, estimate the parameters of that distribution from the sample set, and then use the bootstrap mechanism to draw samples from the assumed distribution, using the so-derived parameters. For example, the next code block does this for a normal distribution.

>>> rv = stats.norm(0,2)

>>> xsamples = rv.rvs(45)

>>> # estimate mean and var from xsamples

>>> mn\_ = np.mean(xsamples)

>>> std\_ = np.std(xsamples)

>>> # bootstrap from assumed normal distribution with

>>> # mn\_,std\_ as parameters

>>> rvb = stats.norm(mn\_,std\_) #plug-in distribution

>>> yboot = rvb.rvs(1000) 3.10 Bootstrapping

201

Recall the sample variance estimator is the following:

1 S 2 = ∑ (X i − ¯X)2 n−1

Assuming that the samples are normally distributed, this means that (n − 1)S 2 /σ2 has a chi-squared distribution with n − 1 degrees of freedom. Thus, the variance, V(S 2 ) = 2σ 4 /(n − 1). Likewise, the MLE plug-in estimate for this is V(S 2 ) = 2ˆ /(n − 1) The following code computes the variance of the sample variance, S2 σ4 using the MLE and bootstrap methods:

>>> # MLE-Plugin Variance of the sample mean

>>> print(2\*(std\_\*\*2)\*\*2/9.) # MLE plugin

2.22670148617726

>>> # Bootstrap variance of the sample mean

>>> print(yboot.var())

3.2946788568183387

>>> # True variance of sample mean

>>> print(2\*(2\*\*2)\*\*2/9.)

3.5555555555555554

This shows that the bootstrap estimate is better here than the MLE plug-in estimate.

Note that this technique becomes even more powerful with multivariate distributions with many parameters because all the mechanics are the same. Thus, the bootstrap is a great all-purpose method for computing standard errors, but, in the limit, is it converging to the correct value? This is the question of consistency. Unfortunately, to answer this question requires more and deeper mathematics than we can get into here. The short answer is that for estimating standard errors, the bootstrap is a consistent estimator in a wide range of cases and so it deﬁnitely belongs in your toolkit.

3.11 Gauss–Markov

In this section, we consider the famous Gauss–Markov problem which will give us an opportunity to use all the material we have so far developed. The GaussMarkov model is the fundamental model for noisy parameter estimation because it estimates unobservable parameters given a noisy indirect measurement. Incarnations of the same model appear in all studies of Gaussian models. This case is an excellent opportunity to use everything we have so far learned about projection and conditional expectation. 202

3

Statistics

Following Luenberger [4] let’s consider the following problem:

y = Wβ +

where W is a n × m matrix, and y is a n × 1 vector. Also, is a n-dimensional normally distributed random vector with zero mean and covariance,

E( T ) = Q

Note that engineering systems usually provide a calibration mode where you can estimate Q so it’s not fantastical to assume you have some knowledge of the noise ˆ statistics. The problem is to ﬁnd a matrix K so that β = K T y approximates β. Note that we only have knowledge of β via y so we can’t measure it directly. Further, note that K is a matrix, not a vector, so there are m × n entries to compute.

We can approach this problem the usual way by trying to solve the MMSE problem:

ˆ min E( ∥ β − β ∥ 2 ) K

which we can write out as

ˆ min E( ∥ β − β ∥ 2 ) = min E( ∥ K T y − β ∥ 2 ) = min E( ∥ K T Wβ + K T − β ∥ 2 ) K K K

and since is the only random variable here, this simpliﬁes to

min ∥ K T Wβ − β ∥ 2 + E( ∥ K T ∥ 2 ) K

The next step is to compute

E( ∥ K T ∥ 2 ) = TrE(K T T K) = Tr(K T QK)

using the properties of the trace of a matrix. We can assemble everything as

min ∥ K T Wβ − β ∥ 2 + Tr(K T QK) K

Now, if we were to solve this for K, it would be a function of β, which is the same ˆ thing as saying that the estimator, β is a function of what we are trying to estimate, β, which makes no sense. However, writing this out tells us that if we had K T W = I, then the ﬁrst term vanishes and the problem simpliﬁes to

min Tr(K T QK) K

with the constraint,

KT W=I 3.11 Gauss–Markov

203

Fig. 3.24 The red circles showthepointstobeestimated in the xy-plane by the black points

This requirement is the same as asserting that the estimator is unbiased,

ˆ E(β) = K T Wβ = β

To line this problem up with our earlier work, let’s consider the i th column of K, k i . Now, we can re-write the problem as

min(k i T Qk i ) k

with

WT k i =ei

and we know how to solve this from our previous work on contrained optimization,

k i = Q −1 W(W T Q −1 W) −1 ei

Now all we have to do is stack these together for the general solution:

K = Q −1 W(W T Q −1 W)−1

It’s easy when you have all of the concepts lined up! For completeness, the covariance of the error is

E(β ˆ − β)(β ˆ − β) T = K T QK = (W T Q −1 W)−1 204

3

Statistics

Fig. 3.25 Focusing on the xyplane in Fig.3.24, the dashed line shows the true value for β versus the mean of the estimated values ⌃ βm

Figure3.24 shows the simulated y data as red circles. The black dots show the ˆ corresponding estimates, β for each sample. The black lines show the true value of β ⌃ versus the average of the estimated β-values, β m . The matrix K maps the red circles in the corresponding dots. Note there are many possible ways to map the red circles to the plane, but the K is the one that minimizes the MSE for β.

Programming Tip The following snippets provide a quick code walkthrough. To simulate the target data, we deﬁne the relevant matrices below:

Q = np.eye(3)\*0.1 # error covariance matrix # this is what we are trying estimate beta = matrix(ones((2,1)))

W = matrix([[1,2],

[2,3], [1,1]])

Then, we generate the noise terms and create the simulated data, y,

ntrials = 50 epsilon = np.random.multivariate\_normal((0,0,0),Q,ntrials).T y=W\*beta+epsilon

Figure3.25 shows more detail in the horizontal xy-plane of Fig.3.24. Figure3.25 ˆ shows the dots, which are individual estimates of β from the corresponding simulated y data. The dashed line is the true value for β and the ﬁlled line ( ⌃ ) is the average βm of all the dots. The gray ellipse provides an error ellipse for the covariance of the estimated β values. 3.11 Gauss–Markov

205

Programming Tip

The following snippets provide a quick walkthrough of the construction of Fig.3.25. To draw the ellipse, we need to import the patch primitive,

from

matplotlib.patches import Ellipse

To compute the parameters of the error ellipse based on the covariance matrix of the individual estimates of β in the bm\_cov variable below,

U,S,V = linalg.svd(bm\_cov) err = np.sqrt((matrix(bm))\*(bm\_cov)\*(matrix(bm).T)) theta = np.arccos(U[0,1])/np.pi\*180

Then, we draw the add the scaled ellipse in the following,

ax.add\_patch(Ellipse(bm,err\*2/np.sqrt(S[0]),

err\*2/np.sqrt(S[1]), angle=theta,color='gray'))

3.12 Nonparametric Methods

So far, we have considered parametric methods that reduce inference or prediction to parameter-ﬁtting. However, for these to work, we had to assume a speciﬁc functional form for the unknown probability distribution of the data. Nonparametric methods eliminate the need to assume a speciﬁc functional form by generalizing to classes of functions.

3.12.1 Kernel Density Estimation

We have already made heavy use of this method with the histogram, which is a special case of kernel density estimation. The histogram can be considered the crudest and most useful nonparametric method that estimates the underlying probability distribution of the data.

Tobeformalandplacethehistogramonthesamefootingasourearlierestimations, suppose that X = [ 0, 1 ] d is the d-dimensional unit cube and that h is the bandwidth or size of a bin or sub-cube. Then, there are N ≈ (1/h) d such bins, each with volume h d , { B 1 , B 2 , . . . , B N } . With all this in place, we can write the histogram has a probability density estimator of the form, 206

3

Statistics

N ˆ θk ph ˆ = ∑ h I (x(x) ∈ B k ) k=1

where

n 1 θ k ˆ = ∑ I (X j ∈ B k ) n j=1

is the fraction of data points (X k ) in each bin, B k . We want to bound the bias and variance of ph ˆ (x). Keep in mind that we are trying to estimate a function of x, but the set of all possible probability distribution functions is extremely large and hard to manage. Thus, we need to restrict our attention to the following class of probability distribution of so-called Lipschitz functions,

P(L) = { p: | p(x) − p(y) | ≤ L ∥ x − y ∥ , ∀ x, y }

Roughly speaking, these are the density functions whose slopes (i.e., growth rates) are bounded by L. It turns out that the bias of the histogram estimator is bounded in the following way:

p(x) − E( ph ˆ (x)) | dx ≤ Lh √ d | ∫

Similarly, the variance is bounded by the following:

C V( ph ˆ (x)) ≤ nhd

for some constant C. Putting these two facts together means that the risk is bounded by

C R(p, p)ˆ = E(p(x) − ph ˆ (x)) 2 dx ≤ L 2 h 2 d + ∫ nhd

This upper bound is minimized by choosing

1 C d+2 h= ( L 2 nd )

In particular, this means that

2 1 d+2 sup R(p, p)ˆ ≤ C0 p∈P(L) ( n )

where the constant C 0 is a function of L. There is a theorem [2] that shows this bound in tight, which basically means that the histogram is a really powerful probability 3.12 Nonparametric Methods

207

2

1 density estimator for Lipschitz functions with risk that goes as ( n ) d+2 . Note that this class of functions is not necessarily smooth because the Lipschitz condition admits non-smooth functions. While this is a reassuring result, we typically do not know which function class (Lipschitz or not) a particular probability belongs to ahead of time. Nonetheless, the rate at which the risk changes with both dimension d and n samples would be hard to understand without this result. Figure3.26 shows the probability distribution function of the β (2, 2) distribution compared to computed histograms for different values of n. The box plots on each of the points show how the variation in each bin of the histogram reduces with increasing n. The risk function R(p, p)ˆ above is based upon integrating the squared difference between the histogram (as a piece-wise function of x) and the probability distribution function.

Programming Tip

The following snippet is the main element of the code for Fig.3.26.

def generate\_samples(n,ntrials=500):

phat = np.zeros((nbins,ntrials)) for k in range(ntrials):

d = rv.rvs(n)

phat[:,k],\_=histogram(d,bins,density=True) return phat

The code uses the histogram function from Numpy. To be consistent with the risk function R(p, ˆp), we have to make sure the bins keyword argument is formatted correctly using a sequence of bin-edges instead of just a single integer. Also, the density=True keyword argument normalizes the histogram appropriately so that the comparison between it and the probability distribution function of the simulated beta distribution is correctly scaled.

3.12.2 Kernel Smoothing

We can extend our methods to other function classes using kernel functions. A one-dimensional smoothing kernel is a smooth function K with the following properties:

K(x)dx = 1 ∫

xK(x)dx = 0 ∫

0 < x 2 K(x)dx < ∞ ∫ 208

3

Statistics

Fig. 3.26 The box plots on each of the points show how the variation in each bin of the histogram reduces with increasing n

For example, K(x) = I (x)/2 is the boxcar kernel, where I (x) = 1 when | x | ≤ 1 and zero otherwise. The kernel density estimator is very similar to the histogram, except now we put a kernel function on every point as in the following:

n 1 1 ∥ x − X i ∥ ˆ = p(x) n ∑ h d K ( h ) i=1

where X ∈ R d . Figure3.27 shows an example of a kernel density estimate using a Gaussian kernel function, K(x) = e −x 2 /2 / √ 2π. There are ﬁve data points shown by the vertical lines in the upper panel. The dotted lines show the individual K(x) function at each of the data points. The lower panel shows the overall kernel density estimate, which is the scaled sum of the upper panel.

There is an important technical result in [2] that states that kernel density estimators are minimax in the sense we discussed in the maximum likelihood Sect.3.4. In broad strokes, this means that the analogous risk for the kernel density estimator is approximately bounded by the following factor:

R(p, p)ˆ n− 2m+d 2m 3.12 Nonparametric Methods

209

Fig. 3.27 The upper panel shows the individual kernel functions placed at each of the data points. The lower panel shows the composite kernel density estimate which is the sum of the individual functions in the upper panel

for some constant C where m is a factor related to bounding the derivatives of the probability density function. For example, if the second derivative of the density function is bounded, then m = 2. This means that the convergence rate for this estimator decreases with increasing dimension d.

Cross-Validation. As a practical matter, the tricky part of the kernel density estimator (which includes the histogram as a special case) is that we need to somehow compute the bandwidth h term using data. There are several rule-of-thumb methods that for some common kernels, including Silverman’s rule and Scott’s rule for Gaussian kernels. For example, Scott’s factor is to simply compute h = n −1/(d + 4) and Silverman’s is h = (n(d + 2)/4) (−1/(d + 4)) . Rules of this kind are derived by assuming the underlying probability density function is of a certain family (e.g., Gaussian), and then deriving the best h for a certain type of kernel density estimator, usually equipped with extra functional properties (say, continuous derivatives of a certain order). In practice, these rules seem to work pretty well, especially for uni-modal probability density functions. Avoiding these kinds of assumptions means computing the bandwidth from data directly and that is where cross-validation comes in.

Cross-validation is a method to estimate the bandwidth from the data itself. The idea is to write out the following integrated squared error (ISE):

ISE( ph ˆ , p) = (p(x) − ph ˆ (x)) 2 dx ∫

= ph ˆ (x) 2 dx − 2 p(x) ph ˆ dx + p(x) 2 dx ∫ ∫ ∫ 210

3

Statistics

The problem with this expression is the middle term,7

p(x) ph ˆ dx ∫

where p(x) is what we are trying to estimate with ph ˆ . The form of the last expression looks like an expectation of ph ˆ over the density of p(x), E( ph ˆ ). The approach is to approximate this with the mean,

n 1 E( ph ˆ ) ≈ ∑ ph ˆ (X i ) n i=1

The problem with this approach is that ph ˆ is computed using the same data that the approximation utilizes. The way to get around this is to split the data into two equally sized chunks D 1 , D 2 ; and then compute ph ˆ for a sequence of different h values over the D 1 set. Then, when we apply the above approximation for the data (Z i ) in the D 2 set,

1 E( ph ˆ ) ≈ ph ˆ (Z i ) | D 2 | Z ∑ 2 i ∈D

Plugging this approximation back into the integrated squared error provides the objective function,

2 ISE ≈ ph ˆ (x) 2 dx − ph ˆ (Z i ) ∫ | D 2 | Z ∑ 2 i ∈D

Some code will make these steps concrete. We will need some tools from Scikit-learn.

>>> from sklearn.model\_selection import train\_test\_split

>>> from sklearn.neighbors.kde import KernelDensity

The train\_test\_split function makes it easy to split and keep track of the D 1 and D 2 sets we need for cross-validation. Scikit-learn already has a powerful and ﬂexible implementation of kernel density estimators. To compute the objective function, we need some basic numerical integration tools from Scipy. For this example, we will generate samples from a β (2, 2) distribution, which is implemented in the stats submodule in Scipy.

>>> from scipy.integrate import quad

>>> from scipy import stats

>>> rv= stats.beta(2,2)

>>> n=100 # number of samples to generate

>>> d = rv.rvs(n)[:,None] # generate samples as column-vector

7 The last term is of no interest because we are only interested in relative changes in the ISE. 3.12 Nonparametric Methods

211

Programming Tip

The use of the [:,None] in the last line formats the Numpy array returned by the rvs function into a Numpy vector with a column dimension of one. This is required by the KernelDensity constructor because the column dimension is used for different features (in general) for Scikit-learn. Thus, even though we only have one feature, we still need to comply with the structured input that Scikit-learn relies upon. There are many ways to inject the additional dimension other than using None. For example, the more cryptic, np.c\_, or the less cryptic [:,np.newaxis] can do the same, as can the np.reshape function.

The next step is to split the data into two halves and loop over each of the h i bandwidths to create a separate kernel density estimator based on the D 1 data,

>>> train,test,\_,\_=train\_test\_split(d,d,test\_size=0.5)

>>> kdes=[KernelDensity(bandwidth=i).fit(train) ... for i in [.05,0.1,0.2,0.3]]

Programming Tip

Note that the single underscore symbol in Python refers to the last evaluated result. The above code unpacks the tuple returned by train\_test\_split into four elements. Because we are only interested in the ﬁrst two, we assign the last two to the underscore symbol. This is a stylistic usage to make it clear to the reader that the last two elements of the tuple are unused. Alternatively, we could assign the last two elements to a pair of dummy variables that we do not use later, but then the reader skimming the code may think that those dummy variables are relevant.

The last step is to loop over the so-created kernel density estimators and compute the objective function.

>>> import numpy as np

>>> for i in kdes:

... f = lambda x: np.exp(i.score\_samples(x)) ... f2 = lambda x: f([[x]])\*\*2 ... print('h=%3.2f\t %3.4f'%(i.bandwidth,quad(f2,0,1)[0] ... -2\*np.mean(f(test)))) ...

h=0.05-1.1323 h=0.10-1.1336 h=0.20-1.1330 h=0.30-1.0810 212

3

Statistics

Fig. 3.28 Each line above is a different kernel density estimator for the given bandwidth as an approximation to the true density function. A plain histogram is imprinted on the bottom for reference

Programming Tip

The lambda functions deﬁned in the last block are necessary because Scikitlearn implements the return value of the kernel density estimator as a logarithm via the score\_samples function. The numerical quadrature function quad from Scipy computes the ∫ ph ˆ (x) 2 dx part of the objective function.

Scikit-learn has many more advanced tools to automate this kind of hyperparameter (i.e., kernel density bandwidth) search. To utilize these advanced tools, we need to format the current problem slightly differently by deﬁning the following wrapper class (Fig.3.28):

>>> class KernelDensityWrapper(KernelDensity):

... def predict(self,x):

... return np.exp(self.score\_samples(x)) ... def score(self,test):

... f = lambda x: self.predict(x) ... f2 = lambda x: f([[x]])\*\*2 ... return -(quad(f2,0,1)[0]-2\*np.mean(f(test))) ...

This is tantamount to reorganizing the above previous code into functions that Scikitlearn requires. Next, we create the dictionary of parameters we want to search over (params) below and then start the grid search with the fit function,

>>> from sklearn.model\_selection import GridSearchCV

>>> params = {'bandwidth':np.linspace(0.01,0.5,10)}

>>> clf = GridSearchCV(KernelDensityWrapper(), param\_grid=params,cv=2)

>>> clf.fit(d) 3.12 Nonparametric Methods

213

GridSearchCV(cv=2,error\_score='raise-deprecating', estimator=KernelDensityWrapper(algorithm='auto',atol=0,bandwidth=1.0, breadth\_first=True,kernel='gaussian',leaf\_size=40, metric='euclidean',metric\_params=None,rtol=0), fit\_params=None,iid='warn',n\_jobs=None, param\_grid={'bandwidth': array([0.01,0.06 444,0.11889,0.17333,0.22778,0.28222,0.33667, 0.39111, 0.44556,0.5])}, pre\_dispatch='2\*n\_jobs',refit=True,return\_train\_score='warn', scoring=None,verbose=0) >>> print (clf.best\_params\_) {'bandwidth': 0.17333333333333334}

The grid search iterates over all the elements in the params dictionary and reports the best bandwidth over that list of parameter values. The cv keyword argument above speciﬁes that we want to split the data into two equally sized sets for training and testing. We can also examine the values of the objective function for each point on the grid as follows:

>>> clf.cv\_results\_['mean\_test\_score'] array([0.60758058,1.06324954,1.11858734,1.13187097,1.12006532, 1.09186225,1.05391076,1.01126161,0.96717292,0.92354959])

Keep in mind that the grid search examines multiple folds for cross-validation to compute the above means and standard deviations. Note that there is also a RandomizedSearchCV in case you would rather specify a distribution of parameters instead of a list. This is particularly useful for searching very large parameter spaces where an exhaustive grid search would be too computationally expensive. Although kernel density estimators are easy to understand and have many attractive analytical properties, they become practically prohibitive for large, high-dimensional datasets.

3.12.3 Nonparametric Regression Estimators

Beyond estimating the underlying probability density, we can use nonparametric methods to compute estimators of the underlying function that is generating the data. Nonparametric regression estimators of the following form are known as linear smoothers:

n yˆ (x) = ∑ i (x)yi i=1

To understand the performance of these smoothers, we can deﬁne the risk as the following:

n 1 R( yˆ , y) = E ∑ ( yˆ (x i ) − y(x i ))2 n ( i=1 ) 214

3

Statistics

and ﬁnd the best yˆ that minimizes this. The problem with this metric is that we do not know y(x), which is why we are trying to approximate it with yˆ (x). We could construct an estimation by using the data at hand as in the following:

n 1 R( ˆ yˆ , y) = ∑ ( yˆ (x i ) − Y i )2 n i=1

where we have substituted the data Y i for the unknown function value, y(x i ). The problem with this approach is that we are using the data to estimate the function and then using the same data to evaluate the risk of doing so. This kind of double-dipping leads to overly optimistic estimators. One way out of this conundrum is to use leaveone-out cross-validation, wherein the yˆ function is estimated using all but one of the data pairs, (X i , Y i ). Then, this missing data element is used to estimate the above risk. Notationally, this is written as the following:

n 1 R( ˆ yˆ , y) = ∑ ( y(−i) ˆ (x i ) − Y i )2 n i=1

where y(−i) ˆ denotes computing the estimator without using the i th data pair. Unfortunately, for anything other than relatively small datasets, it quickly becomes computationally prohibitive to use leave-one-out cross-validation in practice. We’ll get back to this issue shortly, but let’s consider a concrete example of such a nonparametric smoother.

3.12.4 Nearest Neighbors Regression

The simplest possible nonparametric regression method is the k-nearest neighbors regression. This is easier to explain in words than to write out in math. Given an input x, ﬁnd the closest one of the k clusters that contains it and then return the mean of the data values in that cluster. As a univariate example, let’s consider the following chirp waveform:

BW x2 y(x) = cos 2π f o x + ( ( 2τ ))

This waveform is important in high-resolution radar applications. The f o is the start frequency and BW/τ is the frequency slope of the signal. For our example, the fact that it is nonuniform over its domain is important. We can easily create some data by sampling the chirp as in the following:

>>> import numpy as np

>>> from numpy import cos, pi

>>> xi = np.linspace(0,1,100)[:,None] 3.12 Nonparametric Methods

215

Fig. 3.29 The dotted line shows the chirp signal and the solid line shows the nearest neighbor estimate. The gray circles are the sample points that we used to ﬁt the nearest neighbor estimator. The shaded area shows the gaps between the estimator and the unsampled chirp

>>> xin = np.linspace(0,1,12)[:,None]

>>> f0 = 1 # init frequency

>>> BW = 5

>>> y = np.cos(2\*pi\*(f0\*xin+(BW/2.0)\*xin\*\*2))

We can use this data to construct a simple nearest neighbor estimator using Scikitlearn,

>>> from sklearn.neighbors import KNeighborsRegressor

>>> knr=KNeighborsRegressor(2)

>>> knr.fit(xin,y)

KNeighborsRegressor(algorithm='auto',leaf\_size=30,metric='minkowski',

metric\_params=None,n\_jobs=None,n\_neighbors=2,p=2, weights='uniform')

Programming Tip

Scikit-learn has a fantastically consistent interface. The fit function above ﬁts the model parameters to the data. The corresponding predict function returns the output of the model given an arbitrary input. We will spend a lot more time on Scikit-learn in the machine learning chapter. The [:,None] part at the end is just injecting a column dimension into the array in order to satisfy the dimensional requirements of Scikit-learn.

Figure3.29 shows the sampled signal (gray circles) against the values generated by the nearest neighbor estimator (solid line). The dotted line is the full unsampled chirp signal,whichincreasesinfrequencywith x.Thisisimportantforourexamplebecause it adds a nonstationary aspect to this problem in that the function gets progressively 216

3

Statistics

Fig. 3.30 This is the same as Fig.3.29 except that here there are three nearest neighbors used to build the estimator

wigglier with increasing x. The area between the estimated curve and the signal is shaded in gray. Because the nearest neighbor estimator uses only two nearest neighbors, for each new x, it ﬁnds the two adjacent X i that bracket the x in the training data and then averages the corresponding Y i values to compute the estimated value. That is, if you take every adjacent pair of sequential gray circles in the ﬁgure, you ﬁnd that the horizontal solid line splits the pair on the vertical axis. We can adjust the number of nearest neighbors by changing the constructor,

>>> knr=KNeighborsRegressor(3)

>>> knr.fit(xin,y)

KNeighborsRegressor(algorithm='auto',leaf\_size=30,metric='minkowski',

metric\_params=None,n\_jobs=None,n\_neighbors=3,p=2, weights='uniform')

which produces the following corresponding Fig.3.30.

For this example, Fig.3.30 shows that with more nearest neighbors the ﬁt performs poorly, especially toward the end of the signal, where there is increasing variation, because the chirp is not uniformly continuous.

Scikit-learn provides many tools for cross-validation. The following code sets up the tools for leave-one-out cross-validation:

>>> from sklearn.model\_selection import LeaveOneOut

>>> loo=LeaveOneOut()

The LeaveOneOut object is an iterable that produces a set of disjoint indices of the data—one for ﬁtting the model (training set) and one for evaluating the model (testing set). The next block loops over the disjoint sets of training and test indices iterates provided by the loo variable to evaluate the estimated risk, which is accumulated in the out list. 3.12 Nonparametric Methods

217

>>> out=[]

>>> for train\_index, test\_index in loo.split(xin):

... \_=knr.fit(xin[train\_index],y[train\_index]) ... out.append((knr.predict(xi[test\_index])-y[test\_index])\*\*2) ...

>>> print( 'Leave-one-out Estimated Risk: ',np.mean(out),) Leave-one-out Estimated Risk: 1.0351713662681845

The last line in the code above reports leave-one-out’s estimated risk.

Linear smoothers of this type can be rewritten in using the following matrix:

S = [ i (x j ) ]i, j

so that

yˆ = Sy

where y = [Y 1 , Y 2 , . . . , Y n ] ∈ R n and yˆ = [ yˆ (x 1 ), yˆ (x 2 ), . . . , yˆ (x n ) ] ∈ R n . This leads to a quick way to approximate leave-one-out cross-validation as the following:

n 2 ˆ y i − yˆ (x i ) 1 R= n ∑ 1− Si,i i=1 ( )

However, this does not reproduce the approach in the code above because it assumes that each y(−i) ˆ (x i ) is consuming one fewer nearest neighbor than yˆ (x).

We can get this S matrix from the knr object as in the following:

>>> \_= knr.fit(xin,y) # fit on all data

>>> S=(knr.kneighbors\_graph(xin)).todense()/float(knr.n\_neighbors)

The todense part reformats the sparse matrix that is returned into a regular Numpy matrix. The following shows a subsection of this S matrix:

>>> print(S[:5,:5]) [[0.33333333 0.33333333 0.33333333 0. 0. ] [0.33333333 0.33333333 0.33333333 0. 0. ] [0. 0.33333333 0.33333333 0.33333333 0. ] [0. 0. 0.33333333 0.33333333 0.33333333] [0. 0. 0. 0.33333333 0.33333333]]

The sub-blocks show the windows of the the y data that are being processed by the nearest neighbor estimator. For example,

>>> print(np.hstack([knr.predict(xin[:5]),(S\*y)[:5]]))#columns match [[ 0.55781314 0.55781314] [ 0.55781314 0.55781314] [-0.09768138 -0.09768138] [-0.46686876 -0.46686876] [-0.10877633 -0.10877633]] 218

3

Statistics

Or, more concisely checking all entries for approximate equality,

>>> np.allclose(knr.predict(xin),S\*y) True

which shows that the results from the nearest neighbor object and the matrix multiply match.

Programming Tip

Note that because we formatted the returned S as a Numpy matrix, we automatically get the matrix multiplication instead of default element-wise multiplication in the S\*y term.

3.12.5 Kernel Regression

For estimating the probability density, we started with the histogram and moved to the more general kernel density estimate. Likewise, we can also extend regression fromnearestneighborstokernel-basedregressionusingtheNadaraya–Watson kernel regression estimator. Given a bandwidth h > 0, the kernel regression estimator is deﬁned as the following:

K ( x−x h i ) Yi ∑i=1 n yˆ (x) = x−x i ∑ i=1 n K ( h )

Unfortunately, Scikit-learn does not implement this regression estimator; however, Jan Hendrik Metzen makes a compatible version available on github.com.

>>> from kernel\_regression import KernelRegression

This code makes it possible to internally optimize over the bandwidth parameter using leave-one-out cross-validation by specifying a grid of potential bandwidth values (gamma), as in the following:

>>> kr = KernelRegression(gamma=np.linspace(6e3,7e3,500))

>>> kr.fit(xin,y)

KernelRegression(gamma=6000.0,kernel='rbf')

Figure3.31 shows the kernel estimator (heavy black line) using the Gaussian kernel compared to the nearest neighbor estimator (solid light black line). As before, the data points are shown as circles. Figure3.31 shows that the kernel estimator can pick out the sharp peaks that are missed by the nearest neighbor estimator. 3.12 Nonparametric Methods

219

Fig. 3.31 The heavy black line is the Gaussian kernel estimator. The light black line is the nearest neighbor estimator. The data points are shown as gray circles. Note that unlike the nearest neighbor estimator, the Gaussian kernel estimator is able to pick out the sharp peaks in the training data

Thus, the difference between nearest neighbor and kernel estimation is that the latter provides a smooth moving averaging of points whereas the former provides a discontinuous averaging. Note that kernel estimates suffer near the boundaries where there is mismatch between the edges and the kernel function. This problem gets worse in higher dimensions because the data naturally drift toward the boundaries (this is a consequence of the curse of dimensionality). Indeed, it is not possible to simultaneously maintain local accuracy (i.e., low bias) and a generous neighborhood (i.e., low variance). One way to address this problem is to create a local polynomial regression using the kernel function as a window to localize a region of interest. For example,

n x−xi yˆ (x) = ∑ K ( h ) (Y i − α − β x i )2 i=1

and now we have to optimize over the two linear parameters α and β . This method is known as local linear regression [5, 6]. Naturally, this can be extended to higher order polynomials. Note that these methods are not yet implemented in Scikit-learn.

3.12.6 Curse of Dimensionality

The so-called curse of dimensionality occurs as we move into higher and higher dimensions. The term was coined by Bellman in 1961 while he was studying adaptive control processes. Nowadays, the term vaguely refers to anything that becomes more complicated as the number of dimensions increases substantially. Nevertheless, the concept is useful for recognizing and characterizing the practical difﬁculties of highdimensional analysis and estimation.

Consider the volume of an d-dimensional sphere of radius r, 220

3

Statistics

π d/2 rd V s (d,r) = Γ ( d + 1 ) 2

Further, consider the sphere V s (d, 1/2) enclosed by an d-dimensional unit cube. The volume of the cube is always equal to one, but lim d→∞ V s (d, 1/2) = 0. What does this mean? It means that the volume of the cube is pushed away from its center, where the embedded hypersphere lives. Speciﬁcally, the distance from the center of the cube to its vertices in d dimensions is √ d/2, whereas the distance from the center of the inscribing sphere is 1/2. This diagonal distance goes to inﬁnity as d does. For a ﬁxed d, the tiny spherical region at the center of the cube has many long spines attached to it, like a hyper-dimensional sea urchin or porcupine.

Another way to think about this is to consider the > 0 thick peel of the hypersphere,

P = V s (d,r) − V s (d,r − )

Then, we consider the following limit:

Vs (d,r − ) lim P = lim V s (d,r) 1 d→∞ d→∞ ( V s (d,r) )

(3.12.6.1)

r − d = lim V s (d,r) 1 − lim d→∞ d→∞ ( r ) ( )

(3.12.6.2)

= lim V s (d,r) d→∞

(3.12.6.3)

So, in the limit, the volume of the -thick peel consumes the volume of the hypersphere.

What are the consequences of this? For methods that rely on nearest neighbors, exploiting locality to lower bias becomes intractable. For example, suppose we have a d-dimensional space and a point near the origin we want to localize around. To estimate behavior around this point, we need to average the unknown function about this point, but in a high-dimensional space, the chances of ﬁnding neighbors to average are slim. Looked at from the opposing point of view, suppose we have a binary variable, as in the coin-ﬂipping problem. If we have 1000 trials, then, based on our earlier work, we can be conﬁdent about estimating the probability of heads. Now, suppose we have 10 binary variables. Now we have 2 10 = 1024 vertices to estimate. If we had the same 1000 points, then at least 24 vertices would not get any data. To keep the same resolution, we would need 1000 samples at each vertex for a grand total of 1000 × 1024 ≈ 10 6 data points. So, for a tenfold increase in the number of variables, we now have about 1000 more data points to collect to maintain the same statistical resolution. This is the curse of dimensionality.

Perhaps some code will clarify this. The following code generates samples in two dimensions that are plotted as points in Fig.3.32 with the inscribed circle in two dimensions. Note that for d = 2 dimensions, most of the points are contained in the circle. 3.12 Nonparametric Methods

221

Fig. 3.32 Two-dimensional scatter of points randomly and independently uniformly distributed in the unit square. Note that most of the points are contained in the circle. Counter to intuition, this does not persist as the number of dimensions increases

>>> import numpy as np

>>> v=np.random.rand(1000,2)-1/2.

The next code block describes the core computation in Fig.3.33. For each of the dimensions, we create a set of uniformly distributed random variates along each dimension and then compute how close each d-dimensional vector is to the origin. Those that measure one half are those contained in the hypersphere. The histogram of each measurement is shown in the corresponding panel in the Fig.3.33. The dark verticallineshowsthethresholdvalue.Valuestotheleftofthisindicatethepopulation that are contained in the hypersphere. Thus, Fig.3.33 shows that as d increases, fewer points are contained in the inscribed hypersphere. The following code paraphrases the content of Fig.3.33.

fig,ax=subplots() for d in [2,3,5,10,20,50]:

v=np.random.rand(5000,d)-1/2. ax.hist([np.linalg.norm(i) for i in v])

3.12.7 Nonparametric Tests

Determining whether or not two sets of observations derive from the same underlying probability distribution is an important problem. The most popular way to do this is with a standard t-test, but that requires assumptions about normality that may be hard 222

3

Statistics

Fig. 3.33 Each panel shows the histogram of lengths of uniformly distributed d-dimensional random vectors. The population to the left of the dark vertical line are those that are contained in the inscribed hypersphere. This shows that fewer points are contained in the hypersphere with increasing dimension

Fig. 3.34 The black line density function is stochastically larger than the gray one

to justify, which leads to nonparametric methods can get at these questions without such assumptions.

Let V and W be continuous random variables. The variable V is stochastically larger than W if,

P(V ≥ x) ≥ P(W ≥ x)

for all x ∈ R with strict inequality for at least one x. The term stochastically smaller means the obverse of this. For example, the black line density function shown in Fig.3.34 is stochastically larger than the gray one.

The Mann–Whitney–Wilcoxon Test. The Mann–Whitney–Wilcoxon Test approaches the following alternative hypotheses: 3.12 Nonparametric Methods

223

• H 0 : F(x) = G(x) for all x versus

• H a : F(x) ≥ G(x), F stochastically greater than G.

Suppose we have two datasets X and Y and we want to know if they are drawn from the same underlying probability distribution or if one is stochastically greater than the other. There are n x elements in X and n y elements in Y. If we combine these two datasets and rank them, then, under the null hypothesis, any data element should be as likely as any other to be assigned any particular rank. that is, the combined set Z,

Z = { X1 , . . . , X n x , Y1 , . . . , Y n y }

contains n = nx

+ n y elements. Thus, any assignment of n y ranks from the integers n { 1, . . . , n } to { Y 1 , . . . , Y n y } should be equally likely (i.e., P = (n y ) −1 ). Importantly, this property is independent of the F distribution.

That is, we can deﬁne the U statistic as the following:

n x n y U X = ∑ ∑ I(X i ≥ Y j ) i=1 j=1

where I(·) is the usual indicator function. For an interpretation, this counts the number of times that elements of Y outrank elements of X. For example, let us suppose that X = { 1, 3, 4, 5, 6 } and Y = { 2, 7, 8, 10, 11 } . We can get a this in one move using Numpy broadcasting,

>>> import numpy as np

>>> x = np.array([ 1,3,4,5,6 ])

>>> y = np.array([2,7,8,10,11])

>>> U\_X = (y <= x[:,None]).sum()

>>> U\_Y = (x <= y[:,None]).sum()

>>> print (U\_X, U\_Y) 4 21

Note that

n x n y U X + U Y = ∑ ∑ I(Y i ≥ X j ) + I(X i ≥ Y j ) = n x ny i=1 j=1

because I(Yi

≥

X j ) + I(Xi

≥

Y j ) = 1. We can verify this in Python,

>>> print ((U\_X+U\_Y) == len(x)\*len(y)) True

Now that we can compute the U X statistic, we have to characterize it. Let us consider U X . If H 0 is true, then X and Y are identically distributed random variables. Thus all

n x

n y

+

( n x ) allocations of the X-variables in the ordered combined sample are equally likely. Among these, there are ( n x + n n x y −1 ) allocations have a Y variable as the largest 224

3

Statistics

observation in the combined sample. For these, omitting this largest observation does not affect U X because it would not have been counted anyway. The other ( n x n + n −1 y −1 ) x allocations have an element of X as the largest observation. Omitting this observation reduces U X by n y .

With all that, suppose N n x ,n y (u) be the number of allocations of X and Y elements that result in U X = u. Under H 0 situation of equally likely outcomes, we have

Nn x ,n y (u) p n x ,n y (u) = P(U X = u) = n x + n y ( n x )

From our previous discussion, we have the recursive relationship,

N n x ,n y (u) = N n x ,n y −1 (u) + N n x −1,n y (u − n y )

After dividing all of this by ( n x the following:

n y

+ n x

) and using the pn x ,n y

(u) notation above, we obtain

n y nx p n x ,n y (u) = p n x ,n y −1 (u) + p n x −1,n y (u − n y ) n x + n y n x + ny

where0 ≤ u ≤ n x n y .Tostartthisrecursion,weneedthefollowinginitialconditions:

p 0,n y (u x = 0) = 1

p 0,n y (u x > 0) = 0

p n x ,0 (u x = 0) = 1

p n x ,0 (u x > 0) = 0

To see how this works in Python,

>>> def prob(n,m,u):

... if u<0: return 0 ...

if

n==0

or

m==0:

... return int(u==0) ... else:

...

f

=

m/float(m+n)

... return (f\*prob(n,m-1,u) + ... (1-f)\*prob(n-1,m,u-m)) ...

These are shown in Fig.3.35 and approach a normal distribution for large n x , n y , with the following mean and variance: 3.12 Nonparametric Methods

225

Fig. 3.35 The normal approximation to the distribution improves with increasing n x , ny

nx ny E(U) = 2

(3.12.7.1)

nx n y (n x + n y + 1) V(U) = 12

(3.12.7.2)

The variance becomes more complicated when there are ties.

Example. We are trying to determine whether or not one network conﬁguration is faster than another. We obtain the following round-trip times for each of the networks:

>>> X=np.array([ 50.6,31.9,40.5,38.1,39.4,35.1,33.1,36.5,38.7,42.3 ])

>>> Y=np.array([ 28.8,30.1,18.2,38.5,44.2,28.2,32.9,48.8,39.5,30.7 ])

Because there are too few elements to use the scipy.stats.mannwhitneyu function (which internally uses the normal approximation to the U-statistic), we can use our custom function above, but ﬁrst we need to compute the U X statistic using Numpy,

>>> U\_X = (Y <= X[:,None]).sum()

For the p-value, we want to compute the probability that the observed U X statistic at least as great as what was observed, 226

3

Statistics

>>> print(sum(prob(10,10,i) for i in range(U\_X,101))) 0.08274697438784127

This is close to the usual ﬁve percent p-value threshold so it is possible at a slightly higher threshold to conclude that the two sets of samples do not originate from the same underlying distribution. Keep in mind that the usual ﬁve percent threshold is just a guideline. Ultimately, it is up to the analyst to make the call.

Proving Mean and Variance for U-Statistic. To prove Eq.3.12.7.1, we assume there are no ties. One way to get at the result E(U) = n x n y /2,

E(U Y ) = ∑ ∑ P(X i ≤ Y j ) j i

because E(I(X i ≤ Y j )) = P(X i ≤ Y j ). Further, because all the subscripted X and Y variables are drawn independently from the same distribution, we have

E(U Y ) = n x n y P(X ≤ Y)

and also,

P(X ≤ Y) + P(X ≥ Y) = 1

because those are the two mutually exclusive conditions. Because the X variables and Y variables are drawn from the same distribution, we have P(X ≤ Y) = P(X ≥ Y), which means P(X ≤ Y) = 1/2 and therefore E(U Y ) = n x n y /2. Another way to get the same result, is to note that, as we showed earlier, U X + U Y = n x n y . Then, taking the expectation of both sides noting that E(U X ) = E(U Y ) = E(U), gives

2E(U) = n x ny

which gives E(U) = n x n y /2.

Getting the variance is trickier. To start, we compute the following:

E(U X U Y ) = ∑ ∑ ∑ ∑ P(X i ≥ Y j ∧ X k ≤ Y l ) i j k l

Of these terms, we have P(Y j ≤ X i ≤ Y j ) = 0 because these are continuous random variables. Let’s consider the terms of the following type, P(Y i ≤ X k ≤ Y l ). To reduce the notational noise, let’s re-write this as P(Z ≤ X ≤ Y). Writing this out gives

∞ P(Z ≤ X ≤ Y) = (F(Y) − F(Z)) f (y) f (z)dydz ∫ R ∫ Z

where F is the cumulative density function and f is the probability density function (dF(x)/dx = f (x)). Let’s break this up term by term. Using some calculus for the term, 3.12 Nonparametric Methods

227

∞ 1 1 F(Y) f (y)dy = FdF = F(Z)) (1 ∫ Z ∫ F(Z) 2

Then, integrating out the Z variable from this result, we obtain the following:

1 F(Z)2 1 1 f (z)dz = ∫ 2 ( 2 ) 3 R

Next, we compute,

∞ F(Z) f (y)dy f (z)dz = (1 − F(Z))F(Z) f (z)dz ∫ R ∫ Z ∫ R

1 = (1 − F)FdF = ∫ R 6

Finally, assembling the result, we have

1 1 1 P(Z ≤ X ≤ Y) = = 3 6 6

Also, terms like P(X k ≥ Y i ∧ X m ≤ Y i ) = P(X m ≤ Y i ≤ X k ) = 1/6 by the same reasoning. That leaves the terms like P(X k ≥ Y i ∧ X m ≤ Y l ) = 1/4 because of mutual independence and P(X k ≥ Y i ) = 1/2. Now that we have all the terms, we have to assemble the combinatorics to get the ﬁnal answer.

There are n y (n y − 1)n x + n x (n x − 1)n y terms of type P(Y i ≤ X k ≤ Y l ). There are n y (n y −1)n x (n x −1) terms like P(X k ≥ Y i ∧ X m ≤ Y l ). Putting this all together, this means that

nx n y (n x + n y − 2) n y (n x − 1)(n y − 1) nx E(U X U Y ) = + 6 4

To assemble the E(U 2 ) result, we need to appeal to our earlier result,

U X + U Y = nx ny

Squaring both sides of this and taking the expectation gives

E(U X 2 ) + 2E(U X U Y ) + E(U Y 2 ) = n x 2 ny 2

Because E(U X 2 ) = E(U X 2 ) = E(U), we can simplify this as the following:

n x 2 n y 2 − 2E(U X U Y ) E(U 2 ) = 2

nx n y (1 + n x + n y + 3n x n y ) E(U 2 ) = 12 228

3

Statistics

Then, since V(U) = E(U 2 ) − E(U) 2 , we ﬁnally have

nx n y (1 + n x + n y ) = V(U) 12

3.13 Survival Analysis

Survival Curves. The problem is to estimate the length of time units (e.g., subjects, individuals, components) exist in a cohort over time. For example, consider the following data. The rows are the days in a 30-day period and the columns are the individual units. For example, these could be ﬁve patients who all receive a particular treatment on day 0 and then survive (indicated by 1) the next 30 days on not (indicated by 0)

>>> d = pd.DataFrame(index=range(1,8),

... ...

>>> d.loc[3:,'A']=0

>>> d.loc[6:,'B']=0

>>> d.loc[5:,'C']=0

>>> d.loc[4:,'D']=0

columns=['A','B','C','D','E'

data=1)

],

>>> d.index.name='day'

>>> d A B C D E day 1 1 1 1 1 1 2 1 1 1 1 1 3 0 1 1 1 1 4 0 1 1 0 1 5 0 1 0 0 1 6 0 0 0 0 1 7 0 0 0 0 1

Importantly, survival is a one-way street—once a subject is dead, then that subject cannot return to the experiment. This is important because survival analysis is also applied to component failure or other topics where this fact is not so obvious. The following chart shows the survival status of each of the subjects for all seven days. The blue circles indicate that the subject is alive and the red squares indicate death of the subject (Figs.3.36 and 3.37).

There is another important recursive perspective on this calculation. Imagine there is a life raft containing [ A, B, C, D, E ] . Everyone survives until day two when A dies. This leaves four in the life raft [ B, C, D, E ] . Thus, from the perspective of day one, the survival probability is the probability of surviving just up until day two and then surviving day two, P S (t ≥ 2) = P(t ∈ / [ 0, 2) | t < 2)P S (t = 2) = (1)(4/5) = 4/5. 3.13 Survival Analysis

229

Fig. 3.36 The red squares indicate a dead subject, and the blue a living subject

Fig. 3.37 The survival probability decreases by day

In words, this means that surviving past the second day is the product of surviving the second day itself and not having a death up to that point (i.e., surviving up to that point). Using this recursive approach, the survival probability for the third day is P S (t ≥ 3) = P S (t > 3)P S (t = 3) = (4/5)(3/4) = 3/5. Recall that just before the third day, the life raft contains [ B, C, D, E ] and on the third day we have [ B, C, E ] . Thus, from the perspective of just before the third day there are four survivors in the raft and on the third day there are three 3/4. Using this recursive argument generate the same plot and come in handy with censoring.

Censoring and Truncation. Censoring occurs when a subject leaves (right censoring) or enters (left censoring) the study. There are two general types of right censoring. The so-called Type I right censoring is when a subject randomly drops 230

3

Statistics

out of the study. This random drop out is another statistical effect that has to be accounted for in estimating survival. Type II right censoring occurs when the study is terminated when enough speciﬁc random events occur.

Likewise, left censoring occurs when a subject enters the study prior to a certain date, but exactly when this happened is unknown. This happens in study designs involving two separate studies stages. For example, a subject might enroll in the ﬁrst selection process but be ineligible for the second process. Speciﬁcally, suppose a study concerns drug use and certain subjects have used the drug before the study but are unable to report exactly when. These subjects are left censored. Left truncation (a.k.a. staggered entry, delayed entry) is similar except the date of entry is known. For example, a subject that starts taking a drug after being initially left out of the study.

Right censoring is the most common so let’s consider an example. Let’s estimate the survival function given in the following survival times in days:

{ 1, 2, 3 + , 4, 5, 6 + , 7, 8 }

where the censored survival times are indicated by the plus symbol. As before, the survival time at the 0 th day is 8/8 = 1, the ﬁrst day is 7/8, the second day = (7/8)(6/7). Now, we come to the ﬁrst right censored entry. The survival time for the third day is (7/8)(6/7)(5/5) = (7/8)(6/7). Thus, the subject who dropped out is not considered dead and cannot be counted as such but is considered just absent as far as the functional estimation of the probabilities goes. Continuing for the fourth day, we have (7/8)(6/7)(5/5)(4/5), the ﬁfth day, (7/8)(6/7)(5/5)(4/5)(3/4), the sixth (right censored) day (7/8)(6/7)(5/5)(4/5)(3/4)(2/2), and so on. We can summarize this in the following table.

Hazard Functions and Their Properties. Generally, the survival function is a continuous function of time S(t) = P(T > t) where T is the event time (e.g., time of death). Note that the cumulative density function, F(t) = P(T ≤ t) = 1 − S(t) and f (t) = dF(t) dt is the usual probability density function. The so-called hazard function is the instantaneous rate of failure at time t,

f (t) P(T ∈ (t, t + Δt ]| T ≥ t) h(t) = = lim S(t) Δt→0 Δt

Note that is a continuous-limit version of the calculation we performed above. In words, it says given the event time T ≥ t (subject has survived up to t), what is the probability of the event occurring in the differential interval Δt for a vanishingly small Δt. Note that this is not the usual derivative slope from calculus because there is no difference term in the numerator. The hazard function is also called the force of mortality, intensity rate, or the instantaneous risk. Informally, you can think of the hazard function as encapsulating the two issues we are most concerned about: deaths and the population at risk for those deaths. Loosely speaking, the probability density function in the numerator represents the probability of a death occurring in a 3.13 Survival Analysis

231

small differential interval. However, we are not particularly interested in unqualiﬁed deaths, but only deaths that can happen to a speciﬁc at-risk population. Returning to our lifeboat analogy, suppose there are 1000 people in the lifeboat and the probability of anybody falling off the lifeboat is 1/1000. Two things are happening here: (1) the probability of the bad event is small and (2) there are a lot of subjects over which to spread the probability of that bad event. This means that the hazard rate for any particular individual is small. On the other hand, if there are only two subjects in the life raft and the probability of falling off is 3/4, then the hazard rate is high because not only is the unfortunate event probable, the risk of that unfortunate event is shared by only two subjects.

It is a mathematical fact that,

−d log S(t) h(t) = dt

This leads to the following interpretation:

t S(t) = exp − h(u)du := exp(−H(t)) ( ∫ 0 )

where H(t) is the cumulative hazard function. Note that H(t) = − log S(t). Consider a subject whose survival time is 5 years. For this subject to have died at the ﬁfth year, it had to be alive during the fourth year. Thus, the hazard at 5 years is the failure rate per year, conditioned on the fact that the subject survived until the fourth year. Note that this is not the same as the unconditional failure rate per year at the ﬁfth year, because the unconditional rate applies to all units at time zero and does not use information about survival up to that point gleaned from the other units. Thus, the hazard function can be thought of as the point-wise unconditional probability of experiencing the event, scaled by the fraction of survivors up to that point.

3.13.1 Example

To get a sense of this, let’s consider the example where the probability density function is exponential with parameter λ, f (t) = λ exp(−tλ), ∀t > 0. This makes S(t) = 1 − F(t) = exp(−tλ) and then the hazard function becomes h(t) = λ, namely a constant. To see this, recall that the exponential distribution is the only continuous distribution that has no memory:

P(X ≤ u + t | X > u) = 1 − exp(−λt) = P(X ≤ t)

This means no matter how long we have been waiting for a death to occur, the probability of a death from that point onward is the same—thus the hazard function is a constant. 232

3

Statistics

Expectations. Given all these deﬁnitions, it is an exercise in integration by parts to show that the expected life remaining is the following:

∞ E(T ) = S(u)du ∫ 0

This is equivalent to the following:

∞ E(T t = 0) = S(u)du ∫ 0

and we can likewise express the expected remaining life at t as the following:

∞ S(u)du E(T T ≥ t) = ∫ t S(t)

Parametric Regression Models. Because we are interested in how study parameters affect survival, we need a model that can accommodate regression in exogenous (independent) variables (x).

h(t | x) = h o (t) exp(x T β )

where β are the regression coefﬁcients and h o (t) is the baseline instantaneous hazard function. Because the hazard function is always nonnegative, the effects of the covariates enter through the exponential function. These kinds of models are called proportional hazard rate models. If the baseline function is a constant (λ), then this reduces to the exponential regression model given by the following:

h(t | x) = λ exp(x T β )

Cox Proportional Hazards Model. The tricky part about the above proportional hazard rate model is the speciﬁcation of the baseline instantaneous hazard function. In many cases, we are not so interested in the absolute hazard function (or its correctness), but rather a comparison of such hazard functions between two study populations. The Cox model emphasizes this comparison by using a maximum likelihood algorithm for a partial likelihood function. There is a lot to keep track of in this model, so let’s try the mechanics ﬁrst to get a feel for what is going on.

Let j denote the j th failure time, assuming that failure times are sorted in increasing order. The hazard function for subject i at failure time j is h i (t j ). Using the general proportional hazards model, we have

h i (t j ) = h 0 (t j ) exp(z i β ) := h 0 (t j ) ψi

To keep it simple, we have z i ∈ { 0, 1 } that indicates membership in the experimental group (z i = 1) or the control group (z i = 0). Consider the ﬁrst failure time, 3.13 Survival Analysis

233

t 1 for subject i failing is the hazard function h i (t 1 ) = h 0 (t 1 ) ψ i . From the deﬁnitions, the probability that subject i is the one who fails is the following:

h i (t 1 ) h 0 (t 1 ) ψi p1 = = ∑ h k (t 1 ) ∑ h 0 (t 1 ) ψk

where the summation is over all surviving units up to that point. Note that the baseline hazard cancels out and gives the following:

p1 = ψi ∑ k ψk

We can keep computing this for the other failure times to obtain { p 1 , p 2 , . . . p D } . The product of all of these is the partial likelihood, L( ψ ) = p 1 · p 2 · · · p D . The next step is to maximize this partial likelihood (usually logarithm of the partial likelihood) over β . There are a lot of numerical issues to keep track of here. Fortunately, the Python lifelines module can keep this all straight for us.

Let’s see how this works using the Rossi dataset that is available in lifelines.

>>> from lifelines.datasets import load\_rossi

>>> from lifelines import CoxPHFitter, KaplanMeierFitter

>>> rossi\_dataset = load\_rossi()

The Rossi dataset concerns prison recidivism. The fin variable indicates whether or not the subjects received ﬁnancial assistance upon discharge from prison.

• week: week of ﬁrst arrest after release, or censoring time.

• arrest: the event indicator, equal to 1 for those arrested during the period of the study and 0 for those who were not arrested.

• fin: a factor, with levels yes if the individual received ﬁnancial aid after release from prison, and no if he did not; ﬁnancial aid was a randomly assigned factor manipulated by the researchers.

• age: in years at the time of release.

• race: a factor with levels black and other.

• wexp: a factor with levels yes if the individual had full-time work experience prior to incarceration and no if he did not.

• mar: a factor with levels married if the individual was married at the time of release and not married if he was not.

• paro: a factor coded yes if the individual was released on parole and no if he was not.

• prio: number of prior convictions.

• educ: education, a categorical variable coded numerically, with codes 2 (grade 6 or less), 3 (grades 6 through 9), 4 (grades 10 and 11), 5 (grade 12), or 6 (some post-secondary).

• emp1–emp52: factors coded yes if the individual was employed in the corresponding week of the study and no otherwise. 234

3

Statistics

>>> rossi\_dataset.head() week arrest fin age 0 20 1 0 27 1 17 1 0 18 2 25 1 0 19 3 52 0 1 23 4 52 0 0 19

race 1 1 0 1 0

wexp 0 0 1 1 1

mar 0 0 0 1 0

paro 1 1 1 1 1

prio 3 8 13 1 3

Now, we just have to set up the calculation in lifelines, using the scikitlearn style. The lifelines module handles the censoring issues.

>>> cph = CoxPHFitter()

>>> cph.fit(rossi\_dataset,

... duration\_col='week', ... event\_col='arrest') <lifelines.CoxPHFitter: fitted with 432 observations, 318 censored>

>>> cph.print\_summary() # access the results using cph.summary <lifelines.CoxPHFitter: fitted with 432 observations, 318 censored> duration col = 'week' event col = 'arrest' number of subjects = 432 number of events = 114 log-likelihood = -658.75 time fit was run = 2019-03-12 13:54:12 UTC

---

coef exp(coef) fin -0.38 0.68 age -0.06 0.94 race 0.31 1.37 wexp -0.15 0.86 mar -0.43 0.65 paro -0.08 0.92 prio 0.09 1.10 --Concordance = 0.64 Likelihood ratio test = 33.27 on 7 df, -log2(p)=15.37

se(coef) z p

0.19 -1.98 0.05

0.02 -2.61 0.01

0.31 1.02 0.31

0.21 -0.71 0.48

0.38 -1.14 0.26

0.20 -0.43 0.66

0.03 3.19 <0.005

-log2(p) lower 0.95

4.40 -0.75

6.79 -0.10

1.70 -0.29

1.06 -0.57

1.97 -1.18

0.59 -0.47

9.48 0.04

upper 0.95

-0.00

-0.01

0.92

0.27

0.31

0.30

0.15

The values in the summary are plotted in Fig.3.38.

The Cox proportional hazards model object from lifelines allows us to predict the survival function for an individual with given covariates, assuming that the individual just entered the study. For example, for the ﬁrst individual (i.e., row) in the rossi\_dataset, we can use the model to predict the survival function for that individual.

>>> cph.predict\_survival\_function(rossi\_dataset.iloc[0,:]).head() 0 event\_at

0.0 1.000000

1.0 0.997616

2.0 0.995230 3.13 Survival Analysis

235

Fig. 3.38 This shows the ﬁtted coefﬁcients from the summary table for each covariate Fig. 3.39 The Cox proportional hazards model can predict the survival probability for an individual based on their covariates

3.0

4.0

0.992848

0.990468

This result is plotted in Fig.3.39. 236

3

Statistics

References

1. W. Feller, An Introduction to Probability Theory and Its Applications, vol. 1 (Wiley, New York, 1950)

2. L. Wasserman, All of Statistics: A Concise Course in Statistical Inference (Springer, Berlin, 2004)

3. R.A. Maronna, D.R. Martin, V.J. Yohai, Robust Statistics: Theory and Methods. Wiley Series in Probability and Statistics (Wiley, New York, 2006)

4. D.G. Luenberger, Optimization by Vector Space Methods. Professional Series (Wiley, New York, 1968)

5. C. Loader, Local Regression and Likelihood (Springer, Berlin, 2006)

6. T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Series in Statistics (Springer, New York, 2013) Chapter 4

Machine Learnin